Organization TC2100 Bldg./Room PK2
J. S. DEPARTMENT OF COMMERCE
ATENT AND TRADEMARK OFFICE
ASHINGTON, DC 20231
F UNDELIVERABLE RETURN IN TEN DAYS

OFFICIAL BUSINESS

[redacted]

WASHINGTON OCT 03'03 DC
U.S. OFFICIAL MAIL
PENALTY FOR PRIVATE USE $300
PB METER 7349720
$5.75
U.S. POSTAGE

[redacted]

RETURN TO SENDER
Rt. # 1109
Carr. Init.
Date

- Forwarding Order Expired
  Unable To Forward
- ☐ Insufficient Address
- ☐ Moved, Left No Address
- ☐ Unclaimed ☐ Refused
- ☐ Attempted - Not Known
- ☐ No Such Street ☐ Number
- ☐ Vacant ☐ Illegible
- ☐ No Mail Receptacle
- ☐ Box Closed - No Order
- ☐ Returned For Better Address
- ☐ Postage Due

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/605,733	06/28/2000	Yasuaki Yamagishi	SUGI-T0732	6349

7590 11/12/2003

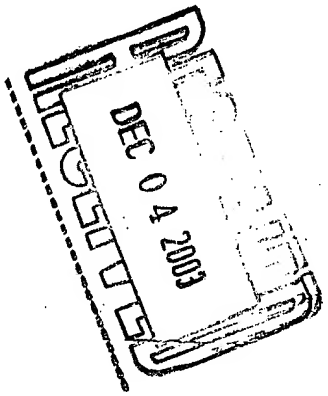
Charles P. Sammut, Esq.
Limbach & Limbach L.L.P.
2001 Ferry Building
San Francisco, CA 94111-4262

EXAMINER	
PHAM, KHANH B	

ART UNIT	PAPER NUMBER
2177	

DATE MAILED: 11/12/2003

Please find below and/or attached an Office communication concerning this application or proceeding.



UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
09/605,733	06/28/2000	Yasuaki Yamagishi	SUGI-T0732	6349

7590 10/03/2003

Charles P. Sammut, Esq.
Limbach & Limbach L.L.P.
2001 Ferry Building
San Francisco, CA 94111-4262

EXAMINER	
PHAM, KHANH B	

ART UNIT	PAPER NUMBER
2177	5

DATE MAILED: 10/03/2003

Please find below and/or attached an Office communication concerning this application or proceeding.

Office Action Summary	Application No.	Applicant(s)	
	09/605,733	YAMAGISHI ET AL.	
	Examiner	Art Unit	
	Khanh B. Pham	2177	

-- *The MAILING DATE of this communication appears on the cover sheet with the correspondence address* --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If the period for reply specified above is less than thirty (30) days, a reply within the statutory minimum of thirty (30) days will be considered timely.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133).
- Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on *28 June 2000* .

2a)☐ This action is **FINAL**. 2b)☒ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-7* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1-7* is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☒ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on *28 June 2000* is/are: a)☐ accepted or b)☒ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

11)☐ The proposed drawing correction filed on _____ is: a)☐ approved b)☐ disapproved by the Examiner.

If approved, corrected drawings are required in reply to this Office action.

12)☐ The oath or declaration is objected to by the Examiner.

Priority under 35 U.S.C. §§ 119 and 120

13)☒ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☒ All b)☐ Some * c)☐ None of:

1.☒ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____ .

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

14)☐ Acknowledgment is made of a claim for domestic priority under 35 U.S.C. § 119(e) (to a provisional application).

a) ☐ The translation of the foreign language provisional application has been received.

15)☐ Acknowledgment is made of a claim for domestic priority under 35 U.S.C. §§ 120 and/or 121.

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☐ Information Disclosure Statement(s) (PTO-1449) Paper No(s) _____ .
4) ☐ Interview Summary (PTO-413) Paper No(s). _____ .
5) ☐ Notice of Informal Patent Application (PTO-152)
6) ☐ Other:

DETAILED ACTION

Priority

1. Receipt is acknowledged of papers submitted under 35 U.S.C. 119(a)-(d), which papers have been placed of record in the file.

2. Should applicant desire to obtain the benefit of foreign priority under 35 U.S.C. 119(a)-(d) prior to declaration of an interference, a translation of the foreign application should be submitted under 37 CFR 1.55 in reply to this action.

Specification

3. Applicant is reminded of the proper language and format for an abstract of the disclosure.

The abstract should be in narrative form and generally limited to a single paragraph on a separate sheet within the range of 50 to 150 words. It is important that the abstract not exceed 150 words in length since the space provided for the abstract on the computer tape used by the printer is limited. The form and legal phraseology often used in patent claims, such as "means" and "said," should be avoided. The abstract should describe the disclosure sufficiently to assist readers in deciding whether there is a need for consulting the full patent text for details.

The language should be clear and concise and should not repeat information given in the title. It should avoid using phrases which can be implied, such as, "The disclosure concerns," "The disclosure defined by this invention," "The disclosure describes," etc.

4. The abstract of the disclosure is objected to because it contains legal term often used in paten claim: "**comprises**" and "**means**". Correction is required. See MPEP § 608.01(b).

5. The title of the invention is not descriptive. A new title is required that is clearly indicative of the invention to which the claims are directed.

Drawings

6. The drawings are objected to because of the following minor informality:

In Fig. 10D, the expression **(2.C, +C)** should have read **(2.C, +c)** because the added entry **c** is a leaf node.

A proposed drawing correction or corrected drawings are required in reply to the Office action to avoid abandonment of the application. The objection to the drawings will not be held in abeyance.

Claim Rejections - 35 USC § 102

7. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (e) the invention was described in (1) an application for patent, published under section 122(b), by another filed in the United States before the invention by the applicant for patent or (2) a patent granted on an application for patent by another filed in the United States before the invention by the applicant for patent, except that an international application filed under the treaty defined in section 351(a) shall have the effects for purposes of this subsection of an application filed in the United States only if the international application designated the United States and was published under Article 21(2) of such treaty in the English language.

8. **Claims 1-7 are rejected under 35 U.S.C. 102(e)** as being anticipated by Prasad et al. (US 5,956,718), hereinafter referred to as "Prasad".

As per claims 1, 3, Prasad teaches a transmitting method and apparatus for transmitting a hierarchical structure of a directory for hierarchically managing locations of contents data, comprising:

- "managing means for managing a hierarchical structure of a directory composed of a container entry and a leaf entry, a container entry containing information in the immediately lower hierarchical level thereof, a leaf entry being disposed in the immediately lower hierarchical level of a container entry, a leaf entry not containing information in the immediately lower hierarchical level thereof" at Col. 2 lines 15-40 and Col. 4 lines 53-67;
- "detecting means for detecting a change of the hierarchical structure of the directory managed by said managing means" at Col. 6 lines 20-30;
- "and obtaining position information and identification information corresponding to the detected result, the position information representing the position of a container entry in the hierarchical structure of the directory, the identification information identifying a leaf entry corresponding to the hierarchical structure of the directory" at Col. 6 lines 1-20;
- "and transmitting means for transmitting the position information and the identification information" at Col. 6 lines 60-65, Col. 15 lines 5-25.

As per claim 2, Prasad teaches the transmitting apparatus as set forth in claim 1, wherein "said detecting means obtains difference information representing a change of the leaf entries corresponding to the detected result of a change of the hierarchical structure of the directory, and wherein said transmitting means transmits the difference information along with the identification information" at Col. 5 line 25 to Col. 6 line 60.

As per claims 4,5, Prasad teaches a receiving apparatus for receiving a hierarchical structure of a directory for hierarchically managing the locations of contents data that are transmitted, comprising:

- "receiving means for receiving position information and identification information, the position information being obtained by detecting a change of container entries, the position information representing the position of a container entry in the hierarchical structure of the directory, the identification information identifying a leaf entry corresponding to the hierarchical structure of the directory" at Col. 5 line 25 to Col. 6 line 60;
- "the directory being composed of container entries and leaf entries, a container entry containing information in the immediately lower hierarchical level thereof, a leaf entry not containing information in the immediately lower hierarchical level thereof" at Col. 4 lines 53-67;
- "obtaining means for selectively obtaining the identification information of a leaf entry in the immediately lower hierarchical level of a container entry represented by the position information corresponding to selection information designated corresponding to the position information" at Col. 6 lines 5-20; and
- "managing means for managing the hierarchical structure of the directory formed with the position information corresponding to the selection information and with the identification information that is selectively obtained" at Col. 2 lines 15-40 and Col. 6 lines 50-60.

As per claims 6, 7, Prasad teaches a transmitting and receiving system for transmitting a hierarchical structure of a directory for hierarchically managing locations of contents data and receiving the transmitted hierarchical structure, comprising:

- "first managing means for managing a hierarchical structure of a directory composed of a container entry and a leaf entry, a container entry containing information in the immediately lower hierarchical level thereof, a leaf entry being disposed in the immediately lower hierarchical level of a container entry, a leaf entry not containing information in the immediately lower hierarchical level thereof" at Col. 2 lines 15-40 and Col. 4 lines 50-67.
- "detecting means for detecting a change of the hierarchical structure of the directory managed by said first managing means and obtaining position information, identification information, and difference information corresponding to the detected result, the position information representing the position of a container entry in the hierarchical structure of the directory, the identification information identifying a leaf entry corresponding to the hierarchical structure of the directory, the difference information representing the difference of leaf entries" at Col. 5 line 25 to Col. 6 line 60;
- "transmitting means for transmitting the position information, the identification information, and the difference information" at Col. 6 lines 30-65, Col. 14 lines 45-55 and Fig. 10;

- "receiving means for receiving the position information, the identification information, and the difference information transmitted by said transmitting means" at Col. 6 lines 30-65, Col. 14 lines 45-55 and Fig. 10;
- "obtaining means for selectively obtaining the identification information of a leaf entry in the immediately lower hierarchical level of a container entry represented by the position information corresponding to selection information designated corresponding to the position information" at Col. 15 lines 5-20;
- "second managing means for managing the hierarchical structure of the directory formed with the position information corresponding to the selection information and with the identification information that is selectively obtained" at Col.15 lines 10-25.

Conclusion

9. The prior art made of record, listed on form PTO-892, and not relied upon, if any, is considered pertinent to applicant's disclosure.

If a reference indicated as being mailed on PTO-FORM 892 has not been enclosed in this action, please contact Lisa Craney whose telephone number is **(703) 305-9601** for faster service.

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Khanh B. Pham whose telephone number is (703) 308-7299. The examiner can normally be reached on Monday through Friday 7:30am to 4:00pm.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, John E Breene can be reached on (703) 305-9790. The fax phone number for the organization where this application or proceeding is assigned is (703) 872-9306.

Any inquiry of a general nature or relating to the status of this application or proceeding should be directed to the receptionist whose telephone number is (703)746-7240.

Khanh B. Pham
Examiner
Art Unit 2177

KBP
September 22, 2003

JEAN R. HOMERE
PRIMARY EXAMINER

HOMERE
EXAMINER

Notice of References Cited	Application/Control No. 09/605,733	Applicant(s)/Patent Under Reexamination YAMAGISHI ET AL.	
	Examiner Khanh B. Pham	Art Unit 2177	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-5,956,718 A	09-1999	Prasad et al.	707/10
	B	US-6,052,724 A	04-2000	Willie et al.	709/223
	C	US-6,065,017 A	05-2000	Barker, Kent D.	707/202
	D	US-6,119,122 A	09-2000	Bunnell, Karl Lee	707/102
	E	US-6,233,623 B1	05-2001	Jeffords et al.	709/316
	F	US-6,466,932 B1	10-2002	Dennis et al.	707/3
	G	US-6,539,381 B1	03-2003	Prasad et al.	707/10
	H	US-6,564,370 B1	05-2003	Hunt, Gary Thomas	717/122
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Daniels et al., "An Algorithm for Replicated Directory",Annual ACM Symposium on Principles of Distributed Computing , Canada, 1983, Pages: 104 - 113.
	V	Balasubramaniam et al., "What is a File Synchronizer", International Conference on Mobile Computing and Networking , 1998, Pages: 98 - 108 .
	W	
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

An algorithm, for replicated directories

Full text Pdf (826 KB)

Source **Annual ACM Symposium on Principles of Distributed Computing** archive
Proceedings of the second annual ACM symposium on Principles of distributed computing table of contents
Montreal, Quebec, Canada
Pages: 104 - 113
Year of Publication: 1983
ISBN:0-89791-110-5

Authors Dean Daniels
Alfred Z. Spector

Sponsors SIGOPS: ACM Special Interest Group on Operating Systems
SIGACT: ACM Special Interest Group on Algorithms and Computation Theory

Publisher ACM Press New York, NY, USA

↑ ABSTRACT

This paper describes a replication algorithm for directory objects based upon Gifford's weighted voting for files. The algorithm associates version number with each possible key on every replica and thereby resolves an ambiguity that arises when directory entries are not stored in every replica. The range of keys associated with a version number changes dynamically; but in all instances, a separate version number is associated with each entry stored on every replica. The algorithm exhibits favorable availability and concurrency properties. There is no performance penalty for associating a version number with every possible key except on Delete operations, and simulation results show this overhead is small.

↑ REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete List rather than only correct and linked references.

1 James E. Allchin, Martin S. McKendry. Object-Based Synchronization and Recovery. Technical Report GIT-CS-82/15, Georgia Institute of Technology, September, 1982.

2 James E. Allchin, Martin S. McKendry. Facilities for Supporting Atomicity in Operating Systems. Technical Report GIT-CS-83/1, Georgia Institute of Technology, January, 1983.

3 Peter A. Alsberg , John D. Day, A principle for resilient sharing of distributed resources, Proceedings of the 2nd international conference on Software engineering, p.562-570, October 13-15,

An Algorithm for Replicated Directories

Dean Daniels and Alfred Z. Spector[1]
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This paper describes a replication algorithm for directory objects based upon Gifford's weighted voting for files. The algorithm associates a version number with each possible key on every replica and thereby resolves an ambiguity that arises when directory entries are not stored in every replica. The range of keys associated with a version number changes dynamically; but in all instances, a separate version number is associated with each entry stored on every replica. The algorithm exhibits favorable availability and concurrency properties. There is no performance penalty for associating a version number with every possible key except on Delete operations, and simulation results show this overhead is small.

CR Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems - *Distributed applications*; D.4.3: [Operating Systems]: File Systems Management - *Directory structures, Distributed file systems*; D.4.5: [Operating Systems]: Reliability - *Fault tolerance*; H.2.4 [Information Systems]: Systems - *Distributed systems*.

General Terms: Algorithms, Reliability.

Additional Key Words and Phrases: Replicated data, Availability, Transaction- based systems.

[1]This work was sponsored in part by: the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or the US government.

1 Introduction

Object replication on distributed computing systems has the goals of increased parallelism, reduced communications costs, and increased resilience to failures. In particular, replication can permit increased *data availability* - continued access to objects despite failures of one or more storage nodes. Unfortunately, it is difficult to achieve increased performance and reliability while ensuring that the semantics of replicated data objects are identical with their non-replicated counterparts.

This paper presents a scheme for replicating directories that permits concurrent operations and arbitrarily high data availability. The semantics of the replicated directory are typical of directories that are stored on a single site. Briefly, directories contain a collection of *entries*, each of which contains a (*key*, *value*) pair with a unique key. The replicated directory has operations similar to the following: Lookup(K:Key) Returns(Boolean, Value), Insert(K:Key, V:Value), Update(K:Key, V:Value), and Delete(K:Key). Trivial modifications of this algorithm may be used to implement sets or similar abstractions.

The replication algorithm that we present is similar to Gifford's weighted voting algorithm [Gifford 79, Gifford 81], and thus, has the same performance and reliability advantages. However, unlike Gifford's algorithm, our algorithm uses a new technique to associate a version number with each possible key at every replica. This technique permits concurrent operations on different entries and solves certain problems in the implementation of the deletion operation. Unlike most replication algorithms, which are concerned with simple objects having only *read* and *write* operations, this algorithm uses the semantic properties of directories, and thereby gains increased performance.

This work on replication is part of a larger research project studying distributed systems that use a transaction facility to support operations on shared abstract data types [Schwarz 82, Spector 83]. The replicated directory described in this paper is an example of a distributed abstract data type whose construction is facilitated by having a flexible underlying transaction mechanism available. Additional components of our research address synchronization, recovery, and communication issues. Groups at MIT and Georgia Institute of Technology are also

investigating the wider use of transactions [Liskov 82, Weihl 83, Allchin 82, Allchin 83].

In the following section of this paper, we survey related replication work and motivate the development of our algorithm. We then describe the algorithm in detail and present performance data that we obtained via simulation. Finally, we discuss additional ways to make the replication algorithm function with greater efficiency and concurrency.

2 Related Work and Motivation

This section discusses the application of existing replication algorithms to the problem of replicated directories, and informally develops the proposed replication strategy. First, unanimous update and primary/secondary copy strategies are briefly discussed. (See Lindsay for a brief survey of these strategies [Lindsay 79].) Then, weighted voting is considered and adapted for use in directory replication.

In the unanimous update strategy, any update operation must be done on all replicas, but reads may be directed to any replica. This replication strategy guarantees data consistency if the systems storing each replica guarantee data consistency locally. Unfortunately, the availability for updates of any object is poor when large numbers of replicas are used. There have been attempts to increase update availability by using the communication system to buffer updates to replicas that are not available. The SDD-1 distributed database system uses an approach like this [Rothnie 77].

In replication strategies based on keeping primary and secondary copies of data, the primary copy receives all updates and then relays the updates to secondary copies. An inquiry may be sent to a secondary copy, but the result may not reflect the most current updates. Because responses to inquiries might not reflect recent updates, it is difficult for a primary/secondary copy replication strategy to duplicate the semantics of a non-replicated object. Techniques for lessening this problem have been developed; for example, the Locus system uses a synchronization site [Popek 81].

Gifford designed a strategy for replication of files, which is based on a scheme called *weighted voting* [Gifford 79, Gifford 81]. This algorithm assigns some number of votes and a version number to each *representative* (or replica) of a replicated *file suite*. Write operations modify each representative in a *write quorum* of W votes and increment the version number of each representative in the quorum. Read operations read from each representative in a *read quorum* of R votes and return data from the representative with the largest version number. The sizes of the read and write quorums are chosen so that R + W is greater than the sum of votes assigned to all representatives. Thus, every read quorum has a non-null intersection with every write quorum and each inquiry is guaranteed to access at least one current copy of the data.

Weighted voting has several attributes that make it particularly appealing as the basis for the design of a replicated directory. First, the sizes of the read and write quorums may be varied to adjust the relative cost and availability of reads and writes. A unanimous update strategy may be specified if desired. Second, representatives with zero votes may be used as hints [Lampson 79]. Third, consistency and recovery are mainly the responsibility of transactional storage systems, which are assumed to hold each representative. Because concurrent operations are synchronized by the transaction system storing each representative, there can be considerable flexibility in the specification and implementation of concurrency control.

While weighted voting is an appealing approach to directory replication, the basic algorithm can not be applied to directories without undesirable concurrency limitations. Even though the semantics of directory operations permit concurrent modifications to different entries, only a single transaction could modify the directory at any time if a directory were stored as a replicated file suite. This is because each representative has a single version number, which causes the serialization of operations that modify the directory.

It might seem that these concurrency limitations could be overcome if each entry in a directory representative were assigned a separate version number. However, with such an approach, representatives might not have a version number for an entry that is stored on other representatives. Because of this, it may not be possible to examine an arbitrary read quorum and determine whether an entry for a particular key exists.

For example, consider a 3-representative directory suite having a read quorum of 2 and a write quorum of 2: we call this a 3-2-2 directory.[1] Initially, each representative in the suite contains entries "a", and "c", and each entry has version number 1 as in Figure 1[2]. Subsequently entry "b" is inserted into representatives A and B with version number 1 (Figure 2). If a "Lookup("b")" request is sent to representatives A and C at this point, representative A will respond with "present with version number 1", and representative C will reply

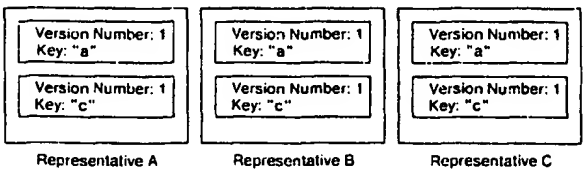


Figure 1: A 3-2-2 Directory Suite - Initial Configuration

[1] The notation x-y-z will refer to a directory having x representatives, a read quorum of y and a write quorum of z. For simplicity, all examples in this paper assume that each representative is assigned one vote.

[2] The *value* field is omitted from all figures to save space.

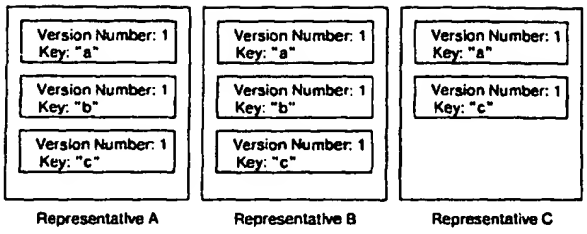


Figure 2: Directory Suite After Inserting "b"

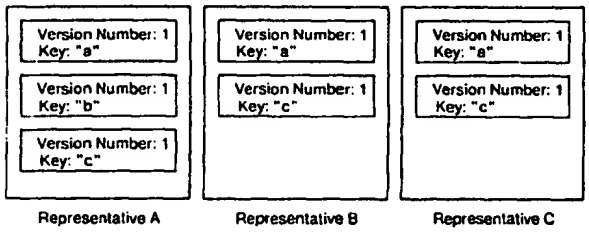


Figure 3: Directory Suite After Deleting "b"

"not present". If entry "b" is then deleted from representatives B and C (Figure 3), "Lookup("b")" requests to representatives A and C will still elicit "present with version number 1", and "not present" responses. Thus, if a directory representative fails to associate a version number with keys for which it has no entry, the responses from a read quorum may not be sufficient to determine if there is an entry in the directory suite for a given key.

The ambiguity demonstrated above is associated with deletions and will not occur if deletions are not permitted. Entries could be updated to indicate that they are "deleted", but the space occupied by "deleted" entries could not easily be reclaimed. An alternative strategy is to eliminate the ambiguity by consulting an additional representative whenever one representative replies "present with version number x" and another representative replies "not present." This approach may be applied to any directory suite configuration, but it results in reduced availability.

As has been demonstrated, associating a version number only with existing entries fails to capture important information about the version numbers of keys for which there are not entries. If, however, a single version number per representative is used, concurrency is limited. A solution is to partition the space of possible keys and to associate a separate version number with each partition.

A directory could be partitioned by placing each key for which there is an entry in a separate partition, and maintaining a single additional partition for all keys that do not have entries. Such a directory keeps a version number with each entry and keeps an additional version number for use with "not present" responses. Under such a partitioning, deletions must increment the "not present" version number. Since the "not present" version number applies to a very large set of keys, this approach suffers from concurrency limitations that are similar to the single version number per representative approach. Alternatively, deletions could be implemented by marking entries to be deleted and then performing a "garbage collection" operation periodically. However, that operation is complex and would itself be a concurrency bottleneck.

This paper will consider partitioning the key space into a set of disjoint ranges by imposing an ordering relation on the keys. The simplest approach is to use a static partitioning; however, the additional concurrency that is achieved might be less than expected. If a small number of ranges were used, then at most that number of transactions could modify a directory concurrently. Also, if transactions modify entries in more than one range, concurrency will be further limited. Even if a large number of ranges were used, an uneven distribution of accesses could limit concurrency.

Below, we concentrate on a technique in which the ranges of keys associated with version numbers change dynamically. A dynamic technique such as this might be desirable for directories having sizes or access patterns that vary widely over time. In this dynamic approach, each directory entry, and, consequently, its key, is in a range by itself with its own version number. Each range of keys between directory entries, called a *gap*, is a separate range with a separate version number.

Because each entry in a directory representative is in a range by itself, lookup operations on such entries return the version number associated with the entry. Lookup operations on keys not in a directory representative return the version number of the gap in which the key appears. Update operations increment the version number of the range containing the entry being updated; insertion operations split a gap;

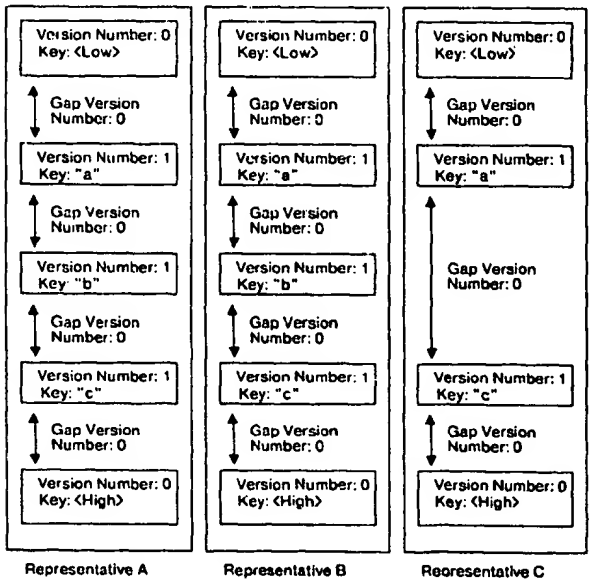


Figure 4: Directory Suite After Inserting "b"

and deletions coalesce the gaps and entries in a range of keys into a single gap. For example, using this approach, entry "b" would be inserted into representatives A and B (of Figure 1) with version number 1, which is one greater than the version number of the gap between "a" and "c" (Figure 4)[3]. If a "Lookup("b")" request were sent to representatives A and C at this point, representative A will respond with "present with version number 1." and representative B will reply "not present with version number 0." Using these responses, a client may determine that there is an entry for "b" since that response has the larger version number. If "b" is subsequently deleted from representatives B and C, then the two gaps on either side of "b" on representative B are coalesced; then on both representatives, the gap between "a" and "c" is assigned version number 2. (Figure 5).

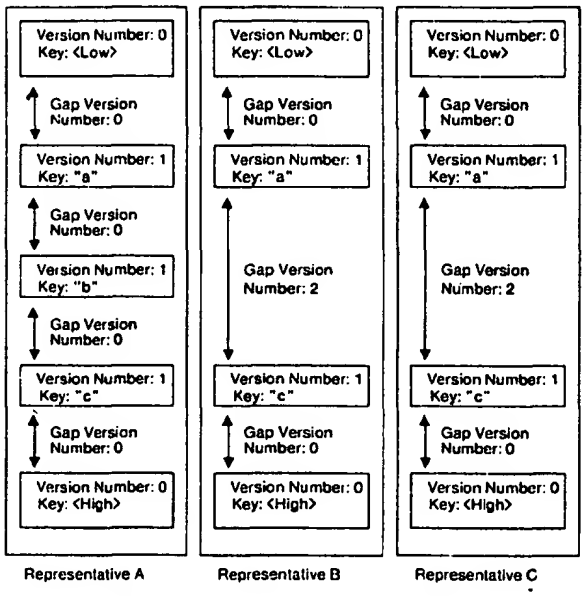


Figure 5: Directory Suite After Deleting "b"

The following section discusses this replication algorithm in more detail.

3 Details of the Algorithm

This section presents the details of the approach to directory replication sketched in the previous section. The descriptions given here are illustrated with program text in a Pascal-like language that allows procedures to return multiple values and includes a remote procedure call primitive. Remote procedure calls are written as "Send(<procedure invocation>) to(<object instance>)" and are assumed to return values in the same fashion as a normal procedure invocation. These remote procedure calls are similar in semantics to those of ARGUS [Liskov 82], except that error responses, such as timeouts, are not considered in these examples. Clarity is emphasized over performance in these descriptions and an inventive reader will find many improvements.

There are three parts to the descriptions given here. First, the operations on directory representatives are identified. Second, the operations on directory suites are described and illustrated and finally, some correctness arguments are given.

3.1 Directory Representatives

In a replicated directory, each directory representative is an instance of an abstract object that stores one copy of the directory data. Arbitrarily complex atomic transactions may be constructed using the basic operations provided by directory representatives. Thus, directory representatives must synchronize concurrent operations performed by different transactions and store critical information in a fashion that recovers from failures. Gifford's weighted voting algorithm makes similar requirements on its file representatives.

Every instance of a directory representative contains two distinguished keys: HIGH and LOW. HIGH is greater than any key that can be inserted into the representative, and LOW is less than any key. HIGH and LOW simplify the directory suite delete operation by ensuring that all keys have a real successor and real predecessor in the directory. *Real predecessor* and *real successor* have an intuitive meaning, but are defined precisely in Section 3.2.

Directory representatives provide typical directory primitives: DirRepLookup and DirRepInsert. In addition, directory representatives provide specialized operations that are used to implement the directory suite deletion operation: DirRepPredecessor, DirRepSuccessor, and DirRepCoalesce. DirRepPredecessor returns the key and version number of the entry in the representative that is the immediate predecessor of the key passed as an argument; it also returns the version number of the gap between the keys. DirRepSuccessor is analogous to DirRepPredecessor. Deletions are performed on a directory representative using the DirRepCoalesce operation, which deletes any entries appearing in a range between two specified entries and assigns a single version number to the resultant gap. Thus, DirRepCoalesce coalesces a range of keys into a single gap. Figure 6 gives sample procedure headings for each of these operations.

Each directory representative must synchronize the concurrent operations of different transactions. While this might be accomplished in many ways, the discussion presented here will assume that type-specific locking is used [Schwarz 82]. In type-specific locking, every operation on an abstract object acquires a lock that is a member of the set of locks associated with that object. A lock compatibility relation is

[3]The directory representatives in Figure 4 contain the special keys LOW and HIGH, which delimit the first and last gaps in the representatives.

used to determine whether a lock may be acquired by a particular transaction.

```
DirRepLookup(x:key)
            Returns(boolean,version,value);
{ If there is an entry for x return TRUE, the
  version number of the entry, and its
  value; otherwise return FALSE and the
  version number of the gap containing x.

  Locks RepLookup(x,x). }

DirRepPredecessor(x:key)
            Returns(key, version, version);
{ Returns the key and version number of the
  entry with the largest key less than x.
  Also returns the version number of the
  gap between x and its predecessor.  There
  need not be an entry for x.

  Locks RepLookup(y,x) where y is the key
  returned. }

DirRepSuccessor(x:key)
             Returns(key,version,version);
{ Returns the key and version number of the
  entry with smallest key greater than x.
  Also returns the version number of the gap
  between x and its successor.  There need
  not be an entry for x.

  Locks RepLookup(x,y) where y is the key
  returned.}

DirRepInsert(x:key,v:version,z:value);
{ Creates an entry for key x with version
  number v and value z.  Updates the entry
  for key x if one already exists.

  Locks RepModify(x,x).}

DirRepCoalesce(l:key,h:key,v:version);
{ Deletes entries for any keys between (but
  not including) l and h.  The resulting gap
  is assigned version number v.  An error is
  indicated if entries do not exist for keys
  l and  h.

  Locks RepModify(l,h). }
```

Figure 6: Directory Representative Operations

The lock classes used in synchronizing a directory representative are the obvious analogs of the lock classes for a single-copy directory (given by Schwarz [Schwarz 82]). However, instead of locking single keys, the lock classes are generalized to lock an entire range of keys and the granting of a lock depends on whether a range of keys to be locked intersects the range of keys already locked by some other transaction. Inquiry operations (DirRepLookup, DirRepPredecessor, and DirRepSuccessor) set RepLookup(σ,τ) locks, where the range of keys explicitly or implicitly accessed by the operation is those keys greater than or equal to σ and less than or equal to τ. A RepModify(σ,τ) lock is obtained on the keys of entries modified by the DirRepInsert and DirRepCoalesce operations.

The lock compatibility relation for operations on directory representatives is illustrated in Figure 7. In the figure, [σ...τ] and [σ'...τ'] are arbitrary non-intersecting ranges of keys, and [σ...τ] and [σ''...τ''] are arbitrary intersecting key ranges. Locks are compatible except that a RepModify lock may not specify a range which intersects the range already specified by another RepModify lock, a RepModify lock may not specify a range which intersects the range already specified by a RepLookup lock, and a RepLookup lock may not specify a range which intersects a range already specified by a RepModify lock. For example, the compatibility relation specifies that a transaction may not be granted a RepModify(σ'',τ'') lock if another transaction already holds a RepModify(σ,τ) lock.

	Lock Held		
Lock Requested	None	RepModify(σ,τ)	RepLookup(σ,τ)
RepModify(σ'',τ'')	OK	No	No
RepModify(σ',τ')	OK	OK	OK
RepLookup(σ'',τ'')	OK	No	OK
RepLookup(σ',τ')	OK	OK	OK

Note: [σ..τ] *intersects* [σ''..τ''] *and* [σ..τ] *does not intersect* [σ'..τ']

Figure 7: Compatibility of Directory Representative Lock Classes

As specified, the lock compatibility relation is sufficiently strong to guarantee that the actions of transactions operating on a directory representative are serializable [Traiger 82], providing that two phase locking is used. This form of synchronization simplifies correctness arguments given in Section 3.3.

3.2 Directory Suites

Directory suites consist of a set of directory representatives, a distribution of votes, and the read and write quorum sizes R and W. Operations on directory representatives are combined to implement a replicated directory based on the weighted voting rules described in Section 2. A Directory suite implements the operations DirSuiteLookup, DirSuiteInsert, DirSuiteUpdate, and DirSuiteDelete.

The DirSuiteLookup operation sends DirRepLookup requests to a read quorum of representatives and returns the results[4] of the reply with the largest version number. Code for this operation is given in Figure 8.

Directory suite modification operations must ensure that the version number of the modified entry is higher than any version number that had been previously associated with the entry's key. In addition, the DirSuiteDelete operation must exercise care so that it does not inadvertently give a higher version number to non-current data.

[4] Figure 8 shows DirSuiteLookup returning a version number as well as a boolean and the value of the entry. The version number is used by the procedures RealPredecessor, DirSuiteInsert, and DirSuiteModify. A user would ignore this number.

```
DirSuiteLookup(x:key)
              Returns(boolean,version,value)

var
   { read quorum has R members }
   quorum : array[1..R] of DirRep;
   v, bestv : version;
   val, bestval : value;
   isin, bestisin : boolean;
   i : integer;

begin
  { collect a read quorum for this operation}
  quorum := CollectReadQuorum();

  bestv := LowestVersion; { a constant }
  { send inquiries to each quorum member }
  for i:= 1 to R do
    begin
      isin,v,val:=Send(DirRepLookup(x))
        to quorum[i];
      if v>bestv then
        begin
          bestv:=v;
          bestval:=val;
          bestisin:=isin;
        end;
    end; { of for i}
  return(bestisin,bestv,bestval);
end; { of DirSuiteLookup }
```

Figure 8: DirSuiteLookup Operation

```
DirSuiteInsert(x:key,z:value);

var
  { write quorum has W members }
  quorum : array[1..W] of DirRep;
  i : integer;
  k : key;
  v : version;
  val : value;
  isin: boolean;

begin
  { first, lookup the key to find the       }
  { current version number                  }
  isin,ver,val:= DirSuiteLookup(x);
  { val ignored }
  if isin then ReportError();

  { find a write quorum }
  quorum := CollectWriteQuorum();

  { The new entry's version number must be }
  { higher than its previous version number }
  { as returned by the DirSuiteLookup call }
  ver:=ver+1;

  { insert the entry in each quorum member }
  for i:= 1 to W do
    Send(DirRepInsert(x,ver,z))
      to(quorum[i]);

end; {of DirSuiteInsert}
```

Figure 9: DirSuiteInsert Operation

The DirSuiteInsert operation is quite simple. DirSuiteInsert first looks up the key to be inserted in a read quorum and uses one greater than the highest version number as the version number for the new entry. The entry is then inserted in a write quorum of representatives. Figure 9 illustrates this operation. The DirSuiteUpdate operation is analogous.

DirSuiteDelete must delete an entry from a write quorum by coalescing a range of keys that includes the entry to be deleted and assigning a higher version number to the resulting gaps. To avoid assigning higher version numbers to data that is not current, the range to be coalesced may not contain directory suite entries other than the one to be deleted. To possess this property, the range must extend from the *real predecessor* of the key to be deleted to its *real successor*. The real predecessor of a key, x, is the entry with the largest key less than x that appears in a write quorum of representatives. The real successor of a key is defined similarly.

Locating the real predecessor and real successor of an entry that is to be deleted is complex. There may be *ghosts* of entries located between the deleted key and its real predecessor or real successor. A ghost is defined as an entry for a key that is no longer present in the directory suite. In addition, the real predecessor or real successor of a key might not be present in some members of the write quorum.

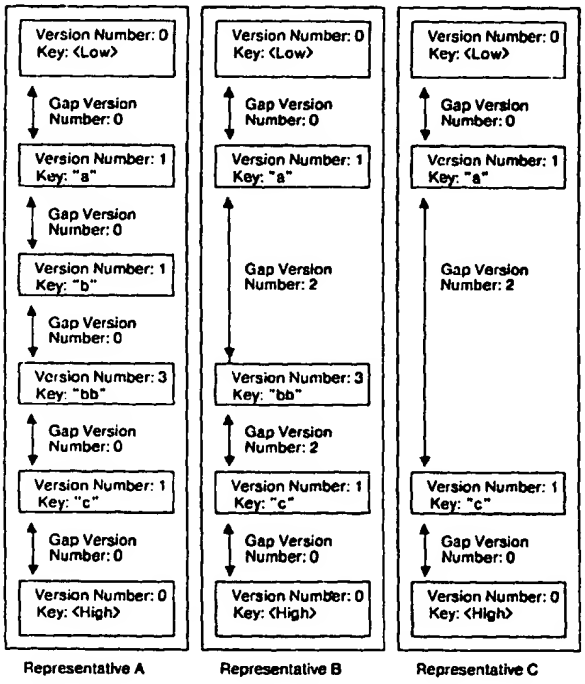


Figure 10: Directory Suite from Figure 5 After Inserting "bb"

These problems are illustrated in Figure 10. In this figure, the real successor of the entry "a" is the entry "bb". However "bb" does not appear in representative C, and the ghost of entry "b" appears between "a" and "bb" in representative A. To delete "a" from representative A and C, the real successor, "bb", must first be located and then copied to representative C. The coalescing of the range from LOW to "bb" eliminates the ghost of entry "b" from representative A, as shown in Figure 11.

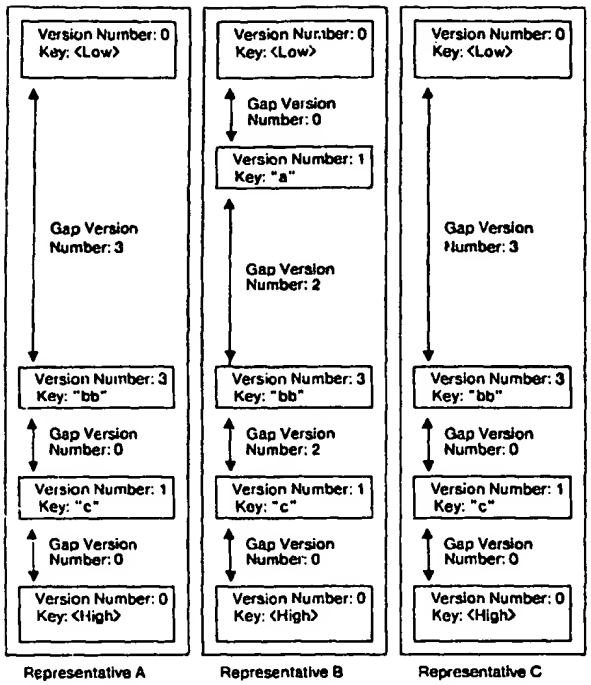


Figure 11: Directory Suite from Figure 10 After Deleting "a"

A straightforward implementation of the procedure RealPredecessor, which locates the real predecessor of a key, is shown in Figure 12. Because of ghost entries, this procedure may have to examine many keys before finding the real predecessor. However, measurements reported in Section 4 indicate that this is not a problem in practice. The DirSuiteDelete operation uses this procedure and the analogous procedure: RealSuccessor. DirSuiteDelete locates the real successor and real predecessor of an entry to be deleted, and inserts entries for the real successor and real predecessor into any member of the write quorum where they do not appear. It then determines the version number to be assigned to the new gap and coalesces the range in each member of the write quorum. DirSuiteDelete is illustrated in Figure 13.

3.3 Correctness Arguments

The correctness of a directory suite's operations depends on DirSuiteLookup always returning current information about a key. Because every read quorum intersects every write quorum, DirSuiteLookup will return current information as long as that information has a version number greater than that of any non-current information and as long as there are no concurrency anomalies. These correctness conditions are the same as those required for Gifford's file replication algorithm.

Two phase locking and the lock compatibility matrices specified in Section 3.1 are strong enough to guarantee the serializability of

```
RealPredecessor(x:key)
  Returns(key,value,version,version);
  {returns key, value, and version number }
  { of x's real predecessor, and the largest}
  { gap version encountered while searching }

var
  {read quorum has R members }
  quorum array[1..R] of DirRep;
  pred, k, pk: key;
  pver, tv, v, vt, maxv: version;
  pvalue: value;
  i: integer;
  isin: boolean;

begin
{ collect a read quorum }
  quorum:=CollectReadQuorum();
  k:=x;
  isin:=false;
  maxv:=LowestVersion; {a constant}
  while not isin do
    begin
      pred:=LowestKey; {a constant }
      for i:=1 to R do
        begin
          pk,tv,v:=Send(DirRepPredecessor(k))
            to(quorum[i]); { tv, ignored }
          pred := Max(pk, pred);
          maxv := Max(v, maxv);
        end; {of for i}
      isin,pver,pvalue:=DirSuiteLookup(pred);
      if not isin then
        k:=pred;
    end; {of while do }
  Return(pred,pvalue,pver,maxv);
end; {of RealPredecessor }
```

Figure 12: RealPredecessor Operation

transactions at any single representative. Traiger et al. [Traiger 82] have shown that if all nodes participating in distributed transaction execution follow two phase locking protocols that guarantee the serializability of transactions at individual nodes, then the resulting global schedule is equivalent to some serial schedule of transactions.

The DirSuiteInsert and DirSuiteUpdate operations both set the version number of the entries they modify to be greater than the greatest version number previously associated with the keys of those entries. Therefore, the current data for each key has a version number greater than that of any non-current data for that key.

DirSuiteDelete coalesces the range between the real predecessor and real successor of the key to be deleted. By the definition of real predecessor and real predecessor, there can be no current entries (other than the entry to be deleted) in the range to be coalesced. The operation assigns to the coalesced range a new version number that is higher than any version number previously associated with every key in that range. Therefore, as with DirSuiteInsert and DirSuiteUpdate, the current data for each key has a version number greater than that of any non-current data for that key.

```
DirSuiteDelete(x:key);

var
  { write quorum has W members }
  quorum : array[1..W] of DirRep;
  i : integer;
  isin: boolean
  succ, pred, k: key;
  pval, sval, val: value;
  pver, sver, v, ver: version;

begin
  { find a write quorum }
  quorum := CollectWriteQuorum();

  { Find the successor of x }
  succ,sval,sver,ver: = RealSuccessor(x);
  { Find the predecessor of x }
  pred,pval,pver,v: = RealPredecessor(x);

  { The version number of the coalesced gap }
  { must be higher than the maximum of any }
  { version numbers in the range coalesced }
  ver := Max(v, ver);
  isin,v,val:=DirSuiteLookup(x);
  {isin, val ignored }
  ver := Max(v, ver);

  { make sure the predecessor and successor}
  { exist in every member of the quorum)
  for i := 1 to W do
    begin
      isin,v,val:= Send(DirRepLookup(succ))
                   to(quorum[i]);
      {v,val ignored}
      if not isin then
        Send(DirRepInsert(succ,sver,svalue))
          to (quorum[i]);
      isin,v,val:= Send(DirRepLookup(pred))
                   to(quorum[i]);
      {v,val ignored}
      if not isin then
        Send(DirRepInsert(pred,pver,pvalue))
          to (quorum[i]);
    end; { for i }

  { coalesce the range in each member }
  for i:= 1 to W do
      Send(DirRepCoalesce(pred,succ,ver+1))
          to (quorum[i]);
end; {of DirSuiteDelete }
```

Figure 13: DirSuiteDelete Operation

4 Performance Characterization

This section presents the results of simulations of this directory replication strategy. There are many statistics that characterize the performance of this algorithm, but only three were selected for the measurements presented here.

The first statistic is labeled "Entries in ranges coalesced" and is the average number of entries (per representative) that lie between the real predecessor and real successor of a deleted key. This statistic counts the entry to be deleted, if it appears in a representative, and any ghosts that may be in the range to be coalesced. Entries for the real predecessors and real successors are not included. This statistic reflects the number of entries that must be examined when the DirSuiteDelete operation is locating the real predecessor and real successor of a entry.

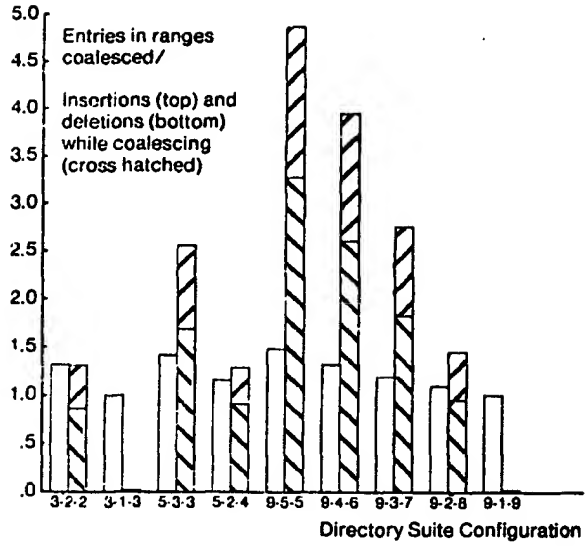


Figure 14: Simulation Results for Various Directory Suites

The second and third statistics, labeled "Insertions while coalescing," and "Deletions while coalescing," are the average numbers of insertions and extra deletions (per suite) performed during each DirSuiteDelete operation. The insertion statistic counts the number of real predecessors and real successors that must be inserted on representatives, and the deletion statistic counts the number of ghost entries that must be deleted. These statistics reflect the extra work done by DirSuiteDelete in addition to the work that would be done by the deletion operation of a unanimous update strategy having the number of replicas in a write quorum.

Figure 14 shows the average results of simulations using directory sizes of approximately one hundred entries with varying numbers of directory representatives and varying sizes of read and write quorums. The duration of each simulation was ten thousand operations, and the members of quorums and the keys to insert, update, or delete were selected randomly from a uniform distribution.

More detailed results for 3-2-2 directories with one hundred, one thousand, and ten thousand entries are shown in Figure 15. The duration of each of these simulations was one hundred thousand operations. The maximums and standard deviations that are shown indicate the statistics do not vary significantly with directory size.[5]

The measurements of the first statistic indicate that the real predecessor and real successor of a key to be deleted will be located quickly if the simulation assumptions hold. For instance, if each member of a read quorum sends the results of three successive

[5] We believe that the statistics for the ten thousand entry directory do not reflect steady state behavior.

100 Entries			1000 Entries			10000 Entries		
Entries in ranges coalesced								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
1.33	9	0.87	1.32	12	0.86	1.20	9	0.76
Deletions while coalescing								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
0.88	8	1.05	0.87	11	1.04	0.67	9	0.90
Insertions while coalescing								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
0.44	2	0.59	0.45	2	0.59	0.53	2	0.64

Figure 15: Detailed Simulation Results for three 3-2-2 Directory Suites

DirRepPredecessor and DirRepSuccessor operations in a single message, the real predecessor and real successor will often be located using one remote procedure call to each member of the quorum. The results for the second and third statistics indicate that the weighted voting algorithm does little extra work during deletions, compared with a unanimous update strategy.

5 Discussion

Though the previous sections motivate and describe the basic replication algorithm, there are many performance issues worthy of mention. First, it is interesting to note that if the memberships of write quorums change infrequently, coalescing during deletions will not be costly. Thus, the statistics presented in the previous section are worse than could be achieved, because quorum members were selected randomly. In some ways, the algorithm behaves similarly to a moving primary update strategy [Alsberg 76] when write quorums change infrequently.

If transactions that operate on a directory exhibit locality of reference with respect to keys, quorums can be chosen that permit reads to be done locally and non-local writes to be distributed among all the non-

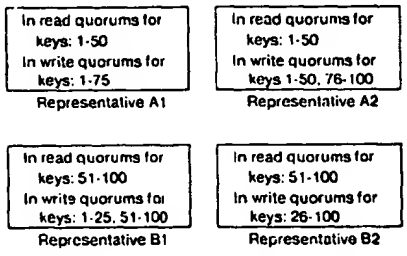


Figure 16: A 4-2-3 Directory Suite Partitioned for Locality

local representatives.[6] For example, consider a 4-2-3 directory suite with key values in the range of 1 to 100, and locality such that transactions of Type A operate on entries having keys 1 to 50, and transactions of Type B operate on entries having keys 51 to 100. We assume that representatives A1 and A2 are local to transactions of Type A and representatives B1 and B2 are local to transactions of Type B. As shown in Figure 16, Type A transactions read from representatives A1 and A2 and direct their updates to A1, A2, and either B1 or B2. Transactions of type B behave similarly. In this example, all inquiries can be done locally and the non-local write that is required for modification operations is evenly distributed among the remote representatives.

With respect to the implementation of the replication algorithm, the sketches we have provided are pedagogically sound, but not the most efficient. Locking rules can be modified to permit greater concurrency without sacrificing serializability. Additionally, inter-representative message traffic can be reduced by combining certain remote procedure calls. We envision that directories could be represented as B-trees [Comer 79]. Version numbers for gaps could be stored in fields in their bounding entries. For some applications, version numbers containing 48 or more bits may be required to prevent version numbers from cycling.

The performance characterizations presented in this paper are based on simulations, however initial work on an analytical treatment indicates that we can obtain similar results from simple analytic models. Further simulations and practical experience are needed in order to quantify the additional concurrency permitted by this directory replication algorithm. We plan to implement this algorithm as well as Gifford's weighted voting algorithm for files using a prototype transaction-based system we are constructing on a modified version of the Accent kernel [Rashid 81].

In summary, this paper has presented a replication algorithm for directories that exhibits favorable performance and availability properties. As is the case with Gifford's algorithm, the exact configuration of suites can be tailored to provide higher or lower availability, and higher or lower performance. This algorithm achieves high concurrency while maintaining consistency by dynamically partitioning the directory by range and associating a version number with each range. Simulation results show the extra costs associated with maintaining the consistency of a directory replicated using our algorithm is low.

Acknowledgments

James Driscoll suggested improvements to our initial dynamic partitioning algorithm that resulted in the algorithm presented in this paper. These improvements simplified the algorithm and reduced the amount of overhead for insert and update operations. Joshua Bloch worked on the analytic model for this algorithm and made other helpful suggestions. Daniel Duchamp, Dave Gifford, Cynthia Hibbard, Robert Sansom, and Peter Schwarz have read and commented on drafts of this paper.

[6] Of course, failures that require the quorums to change will result only in a performance loss.

References

[Allchin 82] James E. Allchin, Martin S. McKendry.
Object-Based Synchronization and Recovery.
Technical Report GIT-CS-82/15, Georgia Institute of Technology, September, 1982.

[Allchin 83] James E. Allchin, Martin S. McKendry.
Facilities for Supporting Atomicity in Operating Systems.
Technical Report GIT-CS-83/1, Georgia Institute of Technology, January, 1983.

[Alsberg 76] P. A. Alsberg, J. D. Day.
A Principle for Resilient Sharing of Distributed Resources.
In *Proc. 2nd International Conf. on Software Engineering*, pages 562-570. October, 1976.

[Comer 79] Douglas Comer.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[Gifford 79] David K. Gifford.
Weighted Voting for Replicated Data.
In *Proc. Seventh Symp. on Operating System Principles*, pages 150-162. ACM, 1979.

[Gifford 81] David K. Gifford.
Information Storage in a Decentralized Computer System.
PhD thesis, Stanford University, 1981.
Available as Xerox Palo Alto Research Center Report CSL-81-8, March 1982.

[Lampson 79] Butler W. Lampson, Robert F. Sproull.
An Open Operating System for a Personal Computer.
In *Proc. Seventh Symp. on Operating System Principles*, pages 98-105. ACM, 1979.

[Lindsay 79] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, Ca., July, 1979.

[Liskov 82] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 7-19.
Albuquerque, NM, January, 1982.

[Popek 81] G. Popek et al.
LOCUS: A Network Transparent, High Reliability Distributed System.
In *Proc. Eighth Symp. on Operating System Principles.*
ACM, 1981.

[Rashid 81] Richard Rashid, George Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. Eighth Symp. on Operating System Principles.*
ACM, 1981.

[Rothnie 77] J. B. Rothnie, N. Goodman, P.A. Bernstein.
The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case).
Technical Report CCA-77-02, Computer Corporation of America, 1977.

[Schwarz 82] Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Carnegie-Mellon Report CMU-CS-82-128, Carnegie-Mellon University, Pittsburgh, PA, September, 1982.

[Spector 83] Alfred Z. Spector, Peter M. Schwarz.
Transactions: A Construct for Reliable Distributed Computing.
Operating Systems Review 17(2):18-35, April, 1983.
Also available as Carnegie-Mellon Report CMU-CS-82-143, January 1983.

[Traiger 82] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, Bruce G. Lindsay.
Transactions and Consistency in Distributed Database Systems.
ACM Transactions on Database Systems 7(3):323-342, September, 1982.

[Weihl 83] W. Weihl, B. Liskov.
Specification and Implementation of Resilient, Atomic Data Types.
In *Symposium on Programming Language Issues in Software Systems.* June, 1983.

What is a file synchronizer?

Full text Pdf (1.21 MB)

Source **International Conference on Mobile Computing and Networking** archive
Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking table of contents
Dallas, Texas, United States
Pages: 98 - 108
Year of Publication: 1998
ISBN:1-58113-035-X

Authors S. Balasubramaniam
Benjamin C. Pierce

Sponsors IEEE-CS : Computer Society
SIGCOMM: ACM Special Interest Group on Data Communication
SIGMOBILE: ACM Special Interest Group on Mobility of Systems, Users, Data and Computing

Publisher ACM Press New York, NY, USA

DOI Bookmark: Use this link to bookmark this Article: http://doi.acm.org/10.1145/288235.288261
What is a DOI?

↑ REFERENCES

Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete List rather than only correct and linked references.

Bri98 Microsoft Windows 95: Vision for mobile computing, 1998. http://www.microsoft.com/windows95/ info/w95mobile.htm.

Dav84 Susan B. Davidson, Optimism and consistency in partitioned distributed database systems, ACM Transactions on Database Systems (TODS), v.9 n.3, p.456-481, Sept. 1984

DDD+94 Dean Daniels , Lip Boon Doo , Alan Downing , Curtis Elsbernd , Gary Hallmark , Sandeep Jain , Bob Jenkins , Peter Lim , Gordon Smith , Benny Souder , Jim Stamos, Oracle's symmetric replication technology and implications for application design, Proceedings of the 1994 ACM SIGMOD international conference on Management of data, p.467, May 24-27, 1994, Minneapolis, Minnesota, United States

DGMS85 Susan B. Davidson , Hector Garcia-Molina , Dale Skeen, Consistency in a partitioned network: a survey, ACM Computing Surveys (CSUR), v.17 n.3, p.341-370, Sept. 1985

DPS+94 Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent

What is a File Synchronizer?

S. Balasubramaniam
Vidam Communications
sundar@vidam.com

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

Abstract

Mobile computing devices intended for disconnected operation, such as laptops and personal organizers, must employ optimistic replication strategies for user files. Unlike traditional distributed systems, such devices do not attempt to present a "single filesystem" semantics: users are aware that their files are replicated, and that updates to one replica will not be seen in another until some point of synchronization is reached (often under the user's explicit control). A variety of tools, collectively called *file synchronizers*, support this mode of operation.

Unfortunately, present-day synchronizers seldom give the user enough information to predict how they will behave under all circumstances. Simple slogans like "Non-conflicting updates are propagated to other replicas" ignore numerous subtleties—e.g., Precisely what constitutes a conflict between updates in different replicas? What does the synchronizer do if updates conflict? What happens when files are renamed? What if the directory structure is reorganized in one replica?

Our goal is to offer a simple, concrete, and precise framework for describing the behavior of file synchronizers. To this end, we divide the synchronization task into two conceptually distinct phases: *update detection* and *reconciliation*. We discuss each phase in detail and develop a straightforward specification of each. We sketch our own prototype implementation of these specifications and discuss how they apply to some existing synchronization tools.

1 Introduction

The growth of mobile computing has brought to fore novel issues in data management, in particular data replication under disconnected operation. Support for replication can be provided either transparently (with filesystem or database support for client-side caching, transaction logs, etc.) or by user-visible tools for explicit replica management. In this paper we investigate one class of user-visible tools—commonly called *file synchronizers*—which allow updates in different replicas to be reconciled at the user's request.

MOBICOM 98 Dallas Texas USA

The overall goal of a file synchronizer is easy to state: it must *detect conflicting updates* and *propagate non-conflicting updates*. However, a good synchronizer is quite tricky to implement. Subtle misunderstandings of the semantics of fileystem operations can cause data to be lost or overwritten. Moreover, the concept of "user update" itself is open to varying interpretations, leading to significant differences in the results of synchronization. Unfortunately, the documentation provided for synchronizers typically makes it difficult to get a clear understanding of what they will do under all circumstances: either there is no description at all or else the description is phrased in terms of low-level mechanisms that do not match the user's intuitive view of the filesystem. In view of the serious damage that can be done by a synchronizer with unintended or unexpected behavior, we would like to establish a concise and rigorous framework in which synchronization can be described and discussed, using terms that both users and implementors can understand.

We concentrate on file synchronization in this paper and only briefly touch upon the finer-grained notion of *data synchronization* offered by newer tools [Puma, DDD+94, etc.], but most of the fundamental issues are the same for file and data synchronization. These issues are also closely related to replication and recovery after partitions in mainstream distributed systems [DGMS85, Kis96, GPJ93, DPS+94, etc.]. Ultimately, we may hope to extend our specification to encompass a wider range of replication mechanisms, from data synchronizers to distributed filesystems and databases.

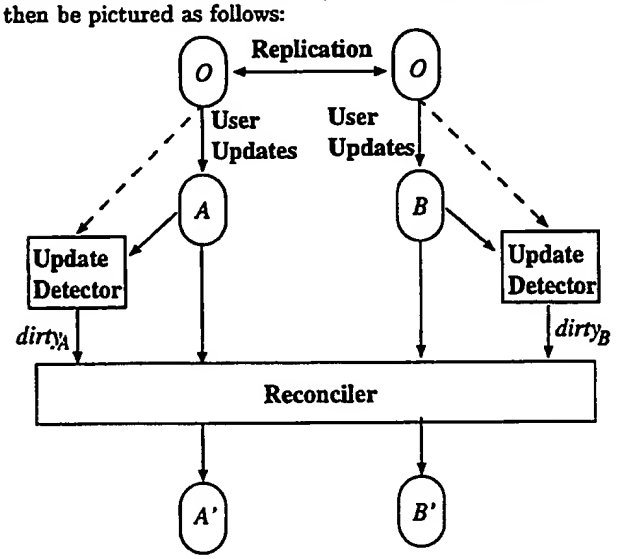
In our model, a file synchronizer is invoked explicitly by an action of the user (issuing a synchronization command, dropping a PDA into a docking cradle, etc.). For purposes of discussion, we identify two cleanly separated phases of the file synchronizer's task: **update detection**—i.e., recognizing where updates have been made to the separate replicas since the last point of synchronization—and **reconciliation**—combining updates to yield the new, synchronized state of each replica.

The update detector for each replica S computes a predicate $dirty_S$ that summarizes the updates that have been made to S. (It is allowed to err on the side of safety, indicating possible updates where none have occurred, but all actual updates must be reported.) The reconciler uses these predicates to decide which replica contains the most up-to-date copy of each file or directory. The contract between the

two components is expressed by the requirement

for all paths p,
$\neg dirty_S(p)$
$\Rightarrow$
$currentContents_S(p) = originalContents_S(p)$,

which the update detector must guarantee and on which the reconciler relies. The whole synchronization process may then be pictured as follows:



The filesystems in both replicas start out with the same contents O. Updates by the user in one or both replicas lead to divergent states A and B at the time when the synchronizer is invoked. The update detectors for the two replicas check the current states of the filesystems (perhaps using some information from O that was stored earlier) and compute update predicates $dirty_A$ and $dirty_B$. The reconciler uses these predicates and the current states A and B to compute new states A' and B', which should coincide unless there were conflicting updates. The specification of the update detector is a relation that must hold between O, A, and $dirty_A$ and between O, B, and $dirty_B$; similarly, the behavior of the reconciler is specified as a relation between A, B, $dirty_A$, $dirty_B$, A', and B'.

The remainder of the paper is organized as follows. We start with some preliminary definitions in Section 2. Then, in Sections 3 and 4, we consider update detection and reconciliation in turn. For update detection, we describe several possible implementation strategies with different performance characteristics. For reconciliation, we first develop a very simple, declarative specification: a small set of natural rules that describe the behavior of a typical synchronizer. We then argue that these rules completely characterize the behavior of any synchronizer satisfying them, and finally show how they can be implemented by a straightforward recursive algorithm. Section 5 sketches our own synchronizer implementation, including the design choices we made in our update detector. Section 6 discusses some existing synchronizers and evaluates how accurately they are described by our specification. Section 7 describes some possible extensions.

Most of our development is independent of the features of particular operating systems and the semantics of their filesystem operations; the one exception is in the implementation of update detectors (Section 3.2), which are necessarily system-specific; our discussion there is biased toward Unix. For the sake of brevity, proofs are omitted.

2 Basic Definitions

To be rigorous about what a synchronizer does to the filesystems it manipulates, the first thing we need is a precise way of talking about the filesystems themselves.

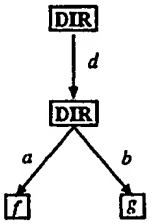
We use the metavariables x and y to range over a set $\mathcal{N}$ of *filenames*. $\mathcal{P}$ is the set of *paths*—finite sequences of names separated by dots. (The dots between path components can be read as slashes by Unix users, backslashes by Windows users, and colons by Mac users.) The metavariables p, q, and r range over paths. The empty path is written ϵ. The concatenation of paths p and q is written $p.q$. We write $|p|$ for the length of path p—i.e., $|\epsilon| = 0$ and $|q.x| = |q| + 1$. We write $q \leq p$ if q is a prefix of p, i.e., if $p = q.r$ for some path r. We write $q < p$ if q is a *proper* prefix of p, i.e., $q \leq p$ and $q \neq p$.

For the purposes of this paper, there is no need to be specific about the contents of individual files. We simply assume that we are given some set $\mathcal{F}$ whose elements are the possible contents of files—for example, $\mathcal{F}$ could be the set of all strings of bytes.

For modeling filesystems, there are many possibilities. Most obviously, we could use the familiar recursive datatype:

$$\mathcal{FS} \quad = \quad \mathcal{F} \uplus (\mathcal{N} \xrightarrow{fin} \mathcal{FS})$$

That is, a "filesystem node" is either a file or a directory, where a file is some $f \in \mathcal{F}$ and a directory is a finite partial function mapping names to nodes of the same form. For example, the filesystem



whose root is a directory containing one subdirectory named d, which contains two files a (with contents f) and b (with contents g), would be represented by the function

$$F = \{d \mapsto D,\\ n \mapsto \perp \text{ for all other names } n\},$$

where $\perp$ marks positions where F is undefined and D is the function

$$D = \{a \mapsto f,\ b \mapsto g,\\ n \mapsto \perp \text{ for all other names } n\}.$$

For purposes of specification, however, it seems more convenient to use a "flat" representation, where a filesystem is a function mapping whole *paths* to their contents. Formally, we say that a filesystem is an element of the set

$$\mathcal{FS} \quad = \quad \{\, S \in \mathcal{P} \xrightarrow{fin} (\mathcal{F} \uplus \mathcal{FS}) \mid \\ \forall p, q \in \mathcal{P}.\ S(p.q) = (S(p))(q) \,\}.$$

of finite partial functions from paths to either files or subfilesystems. The constraint on the second line guarantees that we only consider functions corresponding to tree

structures—i.e., ones where looking up the contents of a composite path $p.q$ yields the same result as first looking up p and then looking up q in the resulting sub-filesystem (where the application expression $(S(p))(q)$ is defined to yield $\perp$ if $S(p)$ is either $\perp$ or a file).

Under this representation, the example filesystem above corresponds to the function

$$F = \{\epsilon \mapsto F,\ d \mapsto D,\ d.a \mapsto f,\ d.b \mapsto g,\\ p \mapsto \perp \text{ for all other paths } p\},$$

where D is the function

$$D = \{\epsilon \mapsto D,\ a \mapsto f,\ b \mapsto g,\\ p \mapsto \perp \text{ for all other paths } p\}.$$

The metavariables O, S, T, A, B, C, and D range over filesystems.

When S is a filesystem, we write $|S|$ for the length of the longest path p such that $S(p) \neq \perp$. We write $children_A(p)$ for the set of names denoting immediate children of path p in filesystem A—that is,

$$children_A(p) = \{q \mid q = p.x \text{ for some } x \wedge A(q) \neq \perp\}.$$

We write $children_{A,B}(p)$ for $children_A(p) \cup children_B(p)$.

We write $isdir_A(p)$ to mean that p refers to a directory (i.e., not a file and not nothing) in the filesystem A. We write $isdir_{A,B}(p)$ iff both $isdir_A(p)$ and $isdir_B(p)$.

To lighten the notation in what follows, we make some simplifying assumptions. First, we assume that, during synchronization, the filesystems are not being modified except by the synchronizer itself. This means that they can be treated as static functions (from paths to contents), as far as the synchronizer is concerned. Second, we assume that, at the end of the previous synchronization, the two filesystems were identical. Third, we handle only two replicas. Finally, we ignore links (both hard and symbolic), file permissions, etc. Section 7 discusses how our development can be refined to relax these restrictions.

3 Update Detection

With these basic definitions in hand, we now turn to the synchronization task itself. This section focuses on update detection, leaving reconciliation for Section 4.

3.1 Specification

We first recapitulate the specification of the update detector sketched in the introduction:

3.1.1 Definition: Suppose O and S are filesystems. Then a predicate $dirty_S$ is said to *(safely) estimate* the updates from O to S if $\neg dirty_S(p)$ implies $O(p) = S(p)$, for all paths p.

Among other things, this definition immediately tells us that, if a given path p is not dirty in either replica, then the two replicas have the same contents at p.

3.1.2 Fact: If A, B, and O are filesystems and $dirty_A$ and $dirty_B$ estimate the updates from O to A and O to B, then $\neg dirty_A(p)$ and $\neg dirty_B(p)$ together imply $A(p) = B(p)$.

One other fact will prove useful in what follows.

3.1.3 Fact: For any filesystem S, $dirty_S$ is up-closed i.e., if $p \leq q$ and $dirty_S(q)$, then $dirty_S(p)$. We shall use this fact to streamline the specification of reconciliation below.

3.2 Implementation Strategies

Update detectors satisfying the above specification can be implemented in many different ways; this section outlines a few and discusses their pragmatic advantages and disadvantages. The discussion is specific to Unix filesystems, but most of the strategies we describe would work with other operating systems too.

3.2.1 Trivial Update Detector

The simplest possible implementation is given by the constantly *true* predicate, which simply marks every file as dirty, with the result that the reconciler must then regard every file (except the ones that happen to be identical in the two filesystems) as a conflict. In some situations, this may actually be an acceptable update detection strategy. On one hand, the fact that the reconciler must actually compare the current contents of all the files in the two filesystems may not be a major issue if the filesystems are small enough and the link between them is fast enough. On the other hand, the fact that all updates lead to conflicts may not be a problem in practice if there are only a few of them. The whole file synchronizer, in this case, degenerates to a kind of recursive remote *diff*.

3.2.2 Exact Update Detector

On the other end of the spectrum is an update detector that computes the *dirty* predicate exactly, for example by keeping a copy of the whole filesystem when it was last synchronized and comparing this state with the current one (i.e., replacing the remote *diff* in the previous case with two local *diffs*).

Detecting updates exactly is expensive, both in terms of disk space and—more importantly—in the time that it takes to compute the difference of the current contents with the saved copies of the filesystem. On the other hand, this strategy may perform well in situations where it is run off-line (in the middle of the night), or where the link between the two computers has very low bandwidth, so that minimizing communication due to false conflicts is critical.

3.2.3 Simple Modtime Update Detector

A much cheaper, but less accurate, update detection strategy involves using the "last modified time" provided by operating systems like Unix. With this strategy, just one value is saved between synchronizations in each replica: the time of the previous synchronization (according to the local clock). To detect updates, each file's last-modified time is compared with this value; if it is older, then the file is not dirty.

Unfortunately, the most naive version of this simple strategy turns out to be wrong. The problem is that, in Unix, renaming a file does not update its modtime, but rather updates the modtime of the directory containing the file: names are a property of directories, not files. For example, suppose we have two files, a and b, and that we move a to b (overwriting b) in one replica. If we examine just the modtime of the path b, we will conclude that it is not dirty, and, in the other replica, a will be deleted without b being changed.

Similarly, it is not enough to look at a file's modtime and its directory's, since the directory itself could have been moved, leaving its modtime alone but changing its parent directory's modtime. To avoid the problem completely, we

must judge a file as dirty if *any* of its ancestors (back to the root of the filesystem) has a modtime more recent than the last synchronization. Unfortunately, this makes the simple modtime detector nearly useless in practice, since any update (file creation, etc.) near the root of the tree leads to large subtrees being marked dirty.

3.2.4 Modtime-Inode Update Detector

A better strategy for update detection under Unix relies on both modtimes and inode numbers. We remember not just the last synchronization time, but also the inode number of every file in each replica. The update detector judges a path as dirty if either (1) its inode number is not the same as the stored one or (2) its modtime is later than the last synchronization time. There is no need to look at the modtimes of any containing directories.

For example, if we move a on top of b, as above, then the new contents of that replica at the path b will be a file with a different inode number than what was there before. Both a and b will be marked as dirty, leading (correctly) to a delete and an update in the other replica.

We have also experimented with a third variant, where inode numbers are stored only for directories, not for each individual file. This uses much less storage than remembering inode numbers for all files, but is not as accurate. Our own experience indicates that storing all the inode numbers is a better tradeoff, on the whole.

3.2.5 On-Line Update Detector

A different kind of update detector—one that is difficult to implement at user level under Unix but possible under some other operating systems such as Windows—requires the ability to observe the complete trace of actions that the user makes to the filesystem. This detector will judge a file to be modified whenever the user has done anything to it (even if the net effect of the user's actions was to return the file to its original state), so it does not, in general, give the same results as the *exact* update detector. But it will normally get close, and may be cheaper to implement than the exact detector.

On-line upate detection presupposes the ability to track all user actions that affect the filesystem; this places it closer to the domain of traditional distributed filesystems (cf., for example, Coda [Kis96, Kum94], Ficus [RHR+94, PJG+97], Bayou [TTP+95, PST+97], and LittleWorks [HH95]).

4 Reconciliation

We now turn our attention to the other major component of the synchronizer, the *reconciler*. We begin by developing a set of simple requirements that any implementation should satisfy (Section 4.1). Then we give a recursive algorithm (Section 4.2) and argue (a) that it satisfies the given requirements, and (b) that the requirements determine its behavior completely, i.e., that any other synchronization algorithm that also satisfies the requirements must be behaviorally indistinguishable from this one (Section 4.3).

4.1 Specification

Suppose that A and B are the current states of two filesystems replicating a common directory structure, and that we have calculated predicates $dirty_A$ and $dirty_B$, estimating the updates in A and B since the last time they were synchronized. Running the reconciler with these inputs will yield new filesystem states C and D. Informally, the behavioral requirements on the synchronizer can be expressed by a pair of slogans: (1) *propagate all non-conflicting updates*, and (2) *if updates conflict, do nothing.*

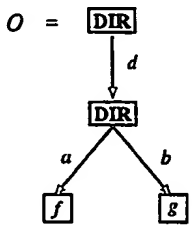
(Of course, an actual synchronization tool will typically try to do better than "do nothing" in the face of conflicting updates: it may, for example, apply additional heuristics based on the types of files involved, ask the user for advice, or allow manual editing on the spot. Such cleanup actions can be incorporated in our model by viewing them as if they had occurred just *before* the synchronizer began its real work.)

We are already committed to a particular formalization of the notion of *update* (cf. Section 3): a path is updated in A if its value in A is different from its original value at the time of last synchronization. We can formalize the notion of *conflicting updates* in an equally straightforward way: updates in A and B are conflicting if the contents of A and B resulting from the updates are different. If A and B are both updated but their new contents happen to agree, these updates will be regarded as non-conflicting. (Another alternative would be to say that overlapping updates always conflict. But this will lead to more false positives in conflict detection.)

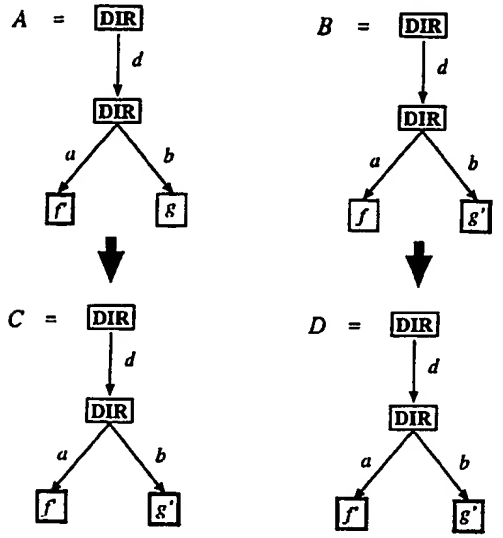
Our specification of the reconciler can be stated as a set of conditions that should hold between the starting states, A and B, and the reconciled states, C and D, for every path p. Informally:

1. If p is not dirty in A, then we know that the entire subtree rooted at p has not been changed in A, and any updates in the corresponding subtree in B should be propagated to both sides; that is, $C(p)$ (the subtree rooted at p in C) and $D(p)$ should be identical to $B(p)$;
2. Conversely, if p is not dirty in B, then we should have $C(p) = D(p) = A(p)$.
3. If p refers to a directory in both A and B, then it should also refer to a directory in C and D. (Note that this requirement makes sense whether or not p is dirty in A or B.)
4. If p is dirty in both A and B and refers to something other than a directory (i.e., it is either a file or $\perp$) in at least one of A and B, then we have potentially conflicting updates. In this case, we should leave things as they are: $C(p) = A(p)$ and $D(p) = B(p)$. (Note that leaving things as they are is the right behavior even in the case where the updates were not actually conflicting—i.e., where it happens that $A(p) = B(p)$.)

A few examples should clarify the consequences of these requirements. Suppose the original state O of the filesystems was

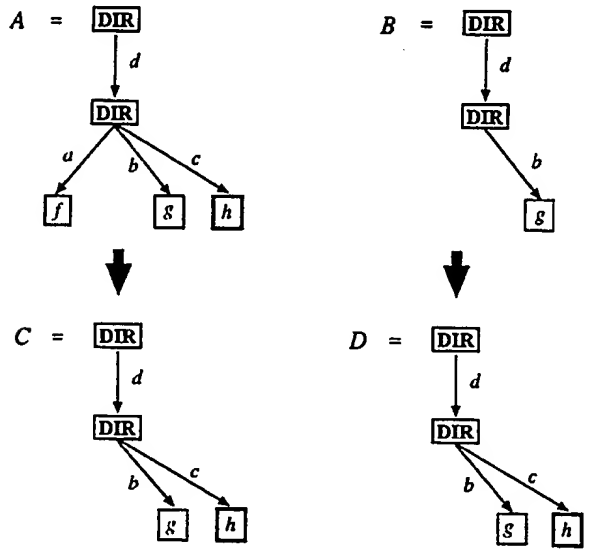


and that we have obtained the current states A and B by modifying the contents of $d.a$ in A and $d.b$ in B. Suppose, furthermore (for the sake of simplicity), that we are using an exact update detector, so that $dirty_A$ is *true* for the paths $d.a$, d, and ϵ and *false* otherwise, while $dirty_B$ is *true* for $d.b$, d, and ϵ. Then, according to the requirements, the resulting states of the two filesystems should be C and D as shown.



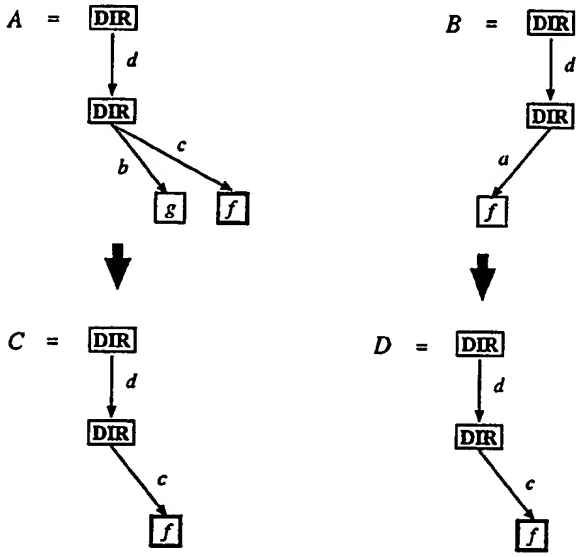
The update in $d.a$ in A has propagated to B and the update in $d.b$ to A, making the final states identical.

Suppose, instead, that the new filesystems A and B are obtained from O by adding a file in A and deleting one in B:

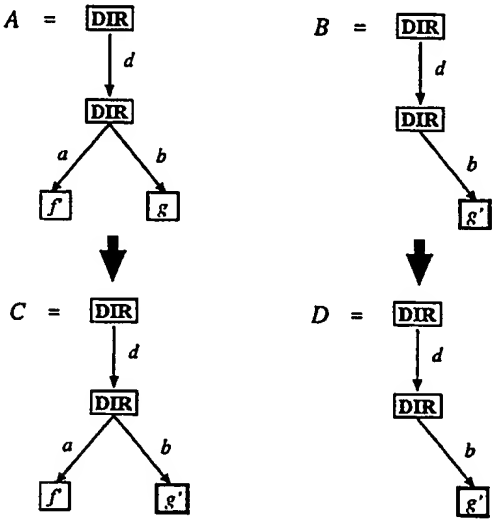


This is an instance of the classic *insert/delete ambiguity* [FM82, GPJ93, PST+97] faced by any synchronization mechanism: if the reconciler could see only the current states A and B, there would be no way for it to know that c had been added in A, as opposed to having existed on both sides originally and having been deleted from B; symmetrically, it could not tell whether a was deleted in B or new in A. The *dirty* predicates provided by the update detector resolve the ambiguity: c is dirty only in A, while a is dirty only in B. (Note that a less accurate update detector might also mark c dirty in B or a dirty in A. The effect would then be a conflict reported by the reconciler and no changes to the filesystems—i.e., the specification requires that synchronization "fail safely.")

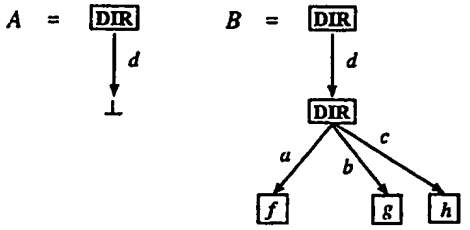
Similarly, suppose the file $d.a$ is renamed, in A, to $d.c$, and that $d.b$ is deleted in B. In A, the paths marked *dirty* are $d.a$, $d.c$, d, and ϵ. In B, the dirty paths are $d.b$, d, and ϵ. So, reconciliation will result in states C and D as shown.



On the other hand, suppose that $d.a$ is modified in A and deleted in B, and that $d.b$ is updated only in B. The dirty paths in A are $d.a$, d, and ϵ; in B they are $d.a$, $d.b$, d, and ϵ. The final clause above thus applies to $d.a$, leaving it unmodified in C and D, while the update to $d.b$ is propagated to A as usual.



One small refinement is needed to complete the specification of reconciliation. In what we've said so far, we've considered *arbitrary* paths p. This is actually slightly too permissive, leading to cases where two of the requirements above make conflicting predictions about the results of synchronization. Suppose, for example, that, A and B are obtained by delete the whole directory d on one side and creating a new file $d.c$ within d on the other:



The contents of $D(d.c)$ should clearly be h after synchronization But what should be the contents of $C(d.c)$? On the one hand, we have $dirty_A(d)$ and $dirty_B(d)$ and $\neg isdir_{A,B}(d)$, so according to the final rule we should have $C(d) = A(d) = \bot$, which implies $C(d.c) = \bot$. But, on the other hand, we have $\neg dirty_A(d.c)$, so according to the first rule, we should have $C(d.c) = B(d.c) = h$.

This is a case of a genuine conflicting update, and we believe the best value for $C(d.c)$ here is $\bot$ (the authors of at least one commercial synchronizer would disagree—cf. Section 6.1). We can resolve the ambiguity by stopping at the first hint of conflict—i.e., by considering only paths p where all the ancestors of p in both A and B refer to directories (and hence do not conflict):

4.1.1 Definition: Let A and B be filesystems. A path p is said to be *relevant* in (A, B) iff $\forall q < p.\ isdir_{A,B}(q)$.

With this refinement, we are ready to state the formal specification of the reconciler.

4.1.2 Definition [Requirements]: The pair of new filesystems (C, D) is said to be a *synchronization* of a pair of original filesystems (A, B) with respect to predicates $dirty_A$ and $dirty_B$ if, for each relevant path p in (A, B), the following conditions are satisfied:

$$\begin{array}{l} \neg dirty_A(p) \\ \quad \Longrightarrow \quad C(p) = D(p) = B(p) \\ \neg dirty_B(p) \\ \quad \Longrightarrow \quad C(p) = D(p) = A(p) \\ isdir_{A,B}(p) \\ \quad \Longrightarrow \quad isdir_{C,D}(p) \\ dirty_A(p) \wedge dirty_B(p) \wedge \neg isdir_{A,B}(p) \\ \quad \Longrightarrow \quad C(p) = A(p) \wedge D(p) = B(p) \end{array}$$

4.2 Algorithm

Having specified the reconciler precisely, we can explore some properties of the specification. In particular, we would like to know that it is *complete*, in the sense that it answers all possible questions about how a reconciler should behave, and that it is *implementable* by a concrete algorithm that terminates on all inputs. We address the latter point first.

For ease of comparison with the abstract requirements above, we present the algorithm in "purely functional" style—as a function taking a pair of filesystems as an argument and returning a fresh pair of filesystems as a result. (Of course, a concrete realization of this algorithm would return no results, performing its task by side-effecting the two filesystems in-place. It should be obvious how to derive such an implementation from the description we give here.)

In the definition, we use the following notation for overwriting part of one filesystem with the contents of the other. Let S and T be functions on paths and p be a path. We write $T \xleftarrow{p} S$ for the function formed by replacing the subtree rooted at p in T with S, defined formally as follows:

$$T \xleftarrow{p} S \quad = \quad \lambda q.\ \text{if } p \leq q \text{ then } S(q) \text{ else } T(q).$$

4.2.1 Definition [Reconciliation Algorithm]: Given predicates $dirty_A$ and $dirty_B$, the algorithm *recon* is defined as follows:

$$\begin{array}{l} recon(A, B, p) = \\ \quad 1)\ \text{if } \neg dirty_A(p) \wedge \neg dirty_B(p) \\ \qquad \text{then } (A, B) \\ \quad 2)\ \text{else if } isdir_{A,B}(p) \\ \qquad \text{then let } \{p_1, p_2, \ldots, p_n\} = children_{A,B}(p) \\ \qquad\qquad \text{(in lexicographic order)} \\ \qquad \text{in let } (A_0, B_0) = (A, B) \\ \qquad\quad \text{let } (A_{i+1}, B_{i+1}) = recon(A_i, B_i, p_{i+1}) \\ \qquad\qquad \text{for } 0 \leq i < n \\ \qquad\quad \text{in } (A_n, B_n) \\ \quad 3)\ \text{else if } \neg dirty_A(p) \\ \qquad \text{then } (A \xleftarrow{p} B, B) \\ \quad 4)\ \text{else if } \neg dirty_B(p) \\ \qquad \text{then } (A, B \xleftarrow{p} A) \\ \quad 5)\ \text{else} \\ \qquad (A, B). \end{array}$$

That is, *recon* takes a pair of filesystems A and B and a path p, and returns a pair of filesystems (C, D) in which the subtrees rooted at p have been synchronized.

An easy induction on $\max(|A|, |B|) - |p|$ shows that *recon* terminates for all filesystems A and B and paths p. Also, observe that updates to the filesystems A and B are performed only through the recursive calls and the grafting function defined above; this ensures that $recon(A, B, p)$ leaves unaffected all parts of A and B that are outside the subtree rooted at p.

4.3 Properties

It remains, now, to verify some properties of the requirements specification and the algorithm. In particular, we can show that (1) the requirements in Definition 4.1.2 fully characterize the behavior of the reconciler; and that (2) the reconciliation algorithm is sound with respect to the specification, i.e., it satisfies the requirements in Definition 4.1.2. It is an immediate consequence of the latter fact that the requirements themselves are consistent, in the sense that, for each A, B, $dirty_A$, and $dirty_B$, there are some C and D such that (C, D) is a synchronization of (A, B) with respect to $dirty_A$ and $dirty_B$.

To facilitate the correctness arguments, we first introduce a refinement of the original requirements that allows us to focus our attention on a specific region of the two filesystems.

4.3.1 Definition: The pair of new filesystems (C, D) is said to be a *synchronization after* p of a pair of original

filesystems (A, B) if p is a relevant path in (A, B) and the following conditions are satisfied for each relevant path $p.q$ in (A, B):

$$\begin{aligned}
&\neg dirty_A(p.q) \\
&\quad \Longrightarrow \quad C(p.q) = D(p.q) = B(p.q) \\
&\neg dirty_B(p.q) \\
&\quad \Longrightarrow \quad C(p.q) = D(p.q) = A(p.q) \\
&isdir_{A,B}(p.q) \\
&\quad \Longrightarrow \quad isdir_{C,D}(p.q) \\
&dirty_A(p.q) \wedge dirty_B(p.q) \wedge \neg isdir_{A,B}(p.q) \\
&\quad \Longrightarrow \quad C(p.q) = A(p.q) \wedge D(p.q) = B(p.q)
\end{aligned}$$

Note that Definition 4.1.2 is just the special case where $p = \epsilon$.

4.3.2 Definition: Paths p and q are *incomparable* if neither is a prefix of the other—i.e., if $p \not\leq q \wedge q \not\leq p$.

4.3.3 Definition: We write $sync_p(C, D, A, B)$ if

1. (C, D) is a synchronization of (A, B) after p,
2. for all paths q, if p and q are incomparable then $C(q) = A(q)$ and $D(q) = B(q)$, and
3. $q \leq p \wedge isdir_{A,B}(q)$ implies $isdir_{C,D}(q)$

The requirements we have placed on the reconciler are *complete* in the sense that they uniquely capture its behavior: given two filesystems which were synchronized at some point in the past, there is at most one pair of new filesystems satisfying the requirements.

4.3.4 Proposition [Uniqueness]: Let A, B, and O be filesystems and suppose that $dirty_A$ and $dirty_B$ estimate the updates from O to A and B respectively. Let p be a relevant path in (A, B). If (C_1, D_1) and (C_2, D_2) are both synchronizations of (A, B) after p, then $C_1(p) = C_2(p)$ and $D_1(p) = D_2(p)$.

Furthermore, the requirements are satisfied by the algorithm.

4.3.5 Proposition [Soundness]: Let A, B, and O be filesystems and suppose that $dirty_A$ and $dirty_B$ estimate the updates from O to A and B respectively. Then $recon(A, B, p) = (C, D)$ implies $sync_p(C, D, A, B)$ for any relevant path p in (A, B).

Together, propositions 4.3.5 and 4.3.4 show that algorithm *recon* is actually *equivalent* to the requirements given in Definition 4.1.2. On the one hand, if $(C, D) = recon(A, B, \epsilon)$, then by soundness we know that (C, D) is a synchronization of A and B. On the other hand, suppose (C, D) is a synchronization of A and B. Since the algorithm is total, it must yield $recon(A, B, \epsilon) = (C', D')$ for some C' and D'. But then by uniqueness, we have $C = C'$ and $D = D'$.

5 Our Implementation

Our main goal has been to understand the synchronization task clearly, not to produce a full-featured synchronizer ourselves. However, we have found it helpful (as well as useful, for our own day to day mobile computing) to experiment with a prototype implementation that straightforwardly embodies the specification we have described.

Our file synchronizer is written in Java, using Java's *Remote Method Invocation* for networking. The design is intended to perform well over both high- and medium-bandwidth links (e.g., ethernet or PPP). To avoid long startup delays, it uses a modtime-inode strategy (cf. Section 3.2.4) for update detection, requiring only minimal summary information to be stored between synchronizations. It operates entirely at user level, without transaction logs or monitor daemons. It currently handles only two replicas at a time and is targeted toward Unix filesystems (though all but the update detector could be used with any operating system, and new update detection modules should be fairly easy to write).

The user interface (see Figure 1) displays all the files in which updates have occurred, using a tree-browser widget; selecting a file from this tree displays its status in a detail dialog at the right and offers a menu of reconciliation options. In the common case where a file has been updated in only one replica, an appropriate action is selected by default and the tree listing shows an arrow indicating which direction the update will be propagated. If both replicas are updated, the tree view displays a question mark, indicating that the user must make some explicit choice. When the user is satisfied, a single button press fires all the selected actions.

Internally, the implementation closely follows the reconciliation algorithm in Section 4.2 (see Figure 2). At the end of every synchronization, a summary of each replica is stored on the disk. The saved information includes the time when each file in the replica was last synchronized and its inode number at that time. At the beginning of the next synchronization, each update detector reads its summary and traverses the file system to detect updates. A file is marked *dirty* if its *ctime*[1] or inode number has changed since the last synchronization. The reconciler then traverses the two replicas in parallel, examining the files for which updates have been detected on either side and posting appropriate records to a tree of pending actions maintained by the user interface.

6 Examples

To explore the utility of our specification, we now discuss some existing synchronizers in terms of the specification framework that we have developed. We do not attempt to provide a complete survey, just a few representative examples.

6.1 Briefcase

Microsoft's *Briefcase* synchronizer [Bri98, Sch96] is part of Windows 95/NT. Its fundamental goals seem to match those embodied in our specification ("propagate updates unless they conflict, in which case do nothing by default")—indeed, even its user interface is fairly similar to our prototype. However, some simple experiments revealed several cases where Briefcase's behavior does not match what is predicted by our specification (or any similar specification that we can think of).

[1] In Unix, a file's *ctime* gets changed if the contents or the attributes (such as permission bits) of the file are changed.

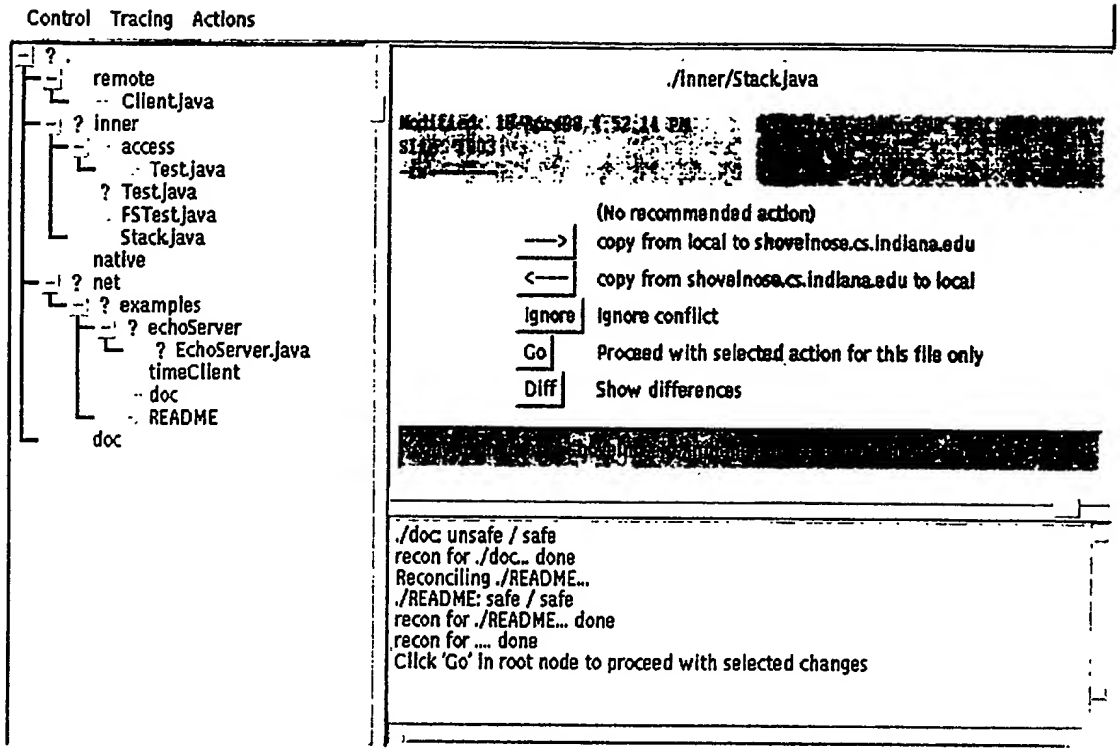


Figure 1: User interface of our synchronizer

The strangest example that we encountered runs as follows. (Since it involves two successive synchronizations, it should be compared with the refined requirements discussed in Section 7.1.) Suppose we have a synchronized filesystem containing a directory (folder) a, a subdirectory $a.b$, and a file $a.b.f$. Now, in one replica, we delete a and all its contents; in the other we modify the contents of $a.b.f$ and add a new subdirectory $a.c$; then we synchronize. At this point, Briefcase reports that no updates are needed. (Strictly speaking, this behavior is correct, since it leaves both replicas unchanged, but a conflict should probably have been reported.) Now, in the second replica, we create a new file $a.b.g$, and synchronize again. This time, the synchronizer does propagate some changes: it recreates a in the first replica, adds subdirectories $a.b$ and $a.c$, and copies $a.b.g$—but not $a.b.f$. Success is reported, but the two filesystems are not identical at the end.

6.2 PowerMerge

According to the manufacturer's advertising [Pow98], the *PowerMerge* synchronizer from Leader Technologies is "used by virtually every large Macintosh organization and is the highest rated file synchronization program on the market today." We tested the "light" version of the program, which is freely downloadable for evaluation.

Although the description of the program's behavior in the user manual again seems to agree with the intentions embodied in our specification, we were unable to make the program behave as documented. For example, deleting a file on one side and then resynchronizing would lead to the file being re-created, not deleted. Also, when both copies of a file have been modified, the most recent copy is propagated, discarding the update in the other copy.

6.3 Rumor

UCLA's Rumor project [Rei97, RPG+96] has built a user-level file synchronizer for Unix filesystems—probably the closest cousin to our own implementation. Although its capabilities go beyond what our specification can describe, Rumor (nearly) satisfies our specification in the two-replica case. (Rumor's model of synchronization originates from the Ficus replicated filesystem; much of our discussion regarding Rumor also applies to the synchronization mechanisms of Ficus [RPG+96, RHR+94, GPJ93].)

In Rumor, reconciliation is performed by a local process in each replica, which works to ensure that the most recent updates to each file in other replicas are eventually reflected in the local state of this replica. For each file in the replica, Rumor maintains a *version vector* reflecting the known updates in all replicas. During reconciliation, this version vector is compared with that of another replica (chosen by the user or determined by availability) to determine which has the latest updates. If the remote copy dominates, then the local copy is modified to reflect the updates; if the local copy dominates, then nothing more is done. (In essence, reconciliation in Rumor uses a "pull model": it is a one-way process.) If there is a conflict, Rumor invokes a *resolver* based on the type of the file; for instance, updates to Unix directories are handled by a "merge resolver" [RHR+94]. Updates eventually get propagated to all replicas by repeated "gossiping" between pairs of replicas.

The update detection strategy in Rumor is a variant of

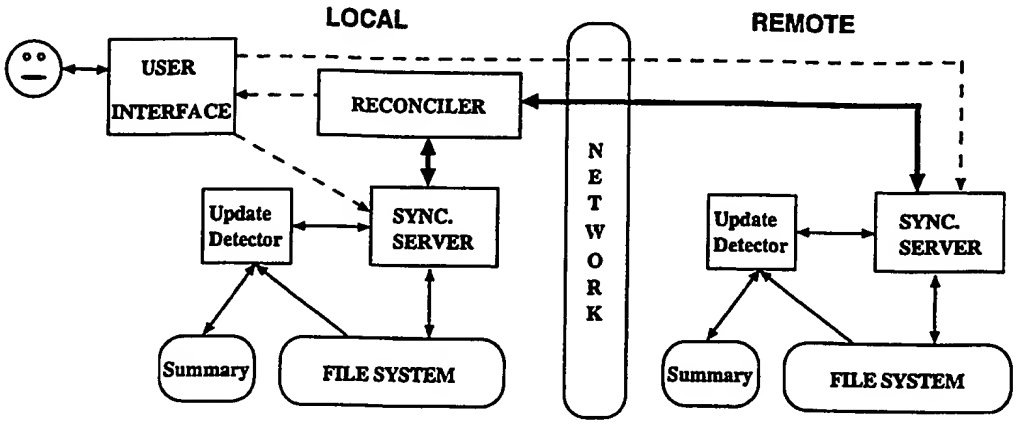


Figure 2: Internals of our synchronizer

the modtime-inode strategy described in Section 3.2.4. Rumor's reconciliation process is more general than that described by our specification. However, it does appear to satisfy our specification if we consider the following special case. (1) There are exactly two Rumor replicas. (2) Both replicas are reconciled at the same time, each treating the other as the source for reconciliation. (3) Overlapping updates are handled by a simple equality check for files (by default, Rumor considers updates to the same file in different replicas as a conflict, even if they result in equal contents) and a recursive merge resolver for directories.

6.4 Distributed Filesystems

Not surprisingly, our model of synchronization has some strong similarities to the replication models underlying mainstream distributed filesystems such as Coda [Kis96, Kum94], Ficus [RHR+94, PJG+97], and Bayou [DPS+94, TTP+95]. Related concepts also have a long history in distributed databases (e.g., [Dav84]).

These systems differ from user-level file synchronizers—and from each other—along numerous dimensions... continuous reconciliation vs. discrete points of synchronization, distinguishing or not between client and server machines, eager vs. lazy reconciliation, use of transaction logs vs. immediate update propagation, etc. Since explicit points of synchronization are not part of the user's conceptual model of these systems, our specification framework is not directly applicable. On the other hand, their underlying concepts of optimistic replication and reconciliation are fundamentally very similar to ours. The intention of synchronization—whenever and however it happens—is (eventually) to propagate nonconflicting updates and to detect and repair conflicting updates. Our specification can therefore be viewed as a first step toward a more general framework in which such systems can be described and compared.

One exception is the system described by Mazer and Tardo [MT94]. Their approach is quite similar to ours in that it includes explicit, user-invoked points of synchronization. Apart from the asymmetry in their setting between clients and servers, our framework could be used to model their system.

6.5 Data Synchronizers

Much of the engineering effort in commercial synchronization tools goes into facilities for *data synchronization*—merging updates to the same file in different replicas using specific knowledge of the structure of the file based on its type (address book, calendar, etc.). Related approaches have long been pursued in distributed database systems [Dav84]) and has resulted in products like Oracle's Symmetric Replication [DDD+94].

Surprisingly, at least some of these tools can be described very directly in our framework. For example, Puma Technology's popular *Intellisync* [Puma, Pumb] can synchronize many kinds of databases between handheld PDAs, laptop computers, and workstations. It requires that one or more *key fields* be chosen for each type of database to be synchronized. (For example, in an address book the key fields might be the first and last name; in a calendar database they could be the date, time, and description of an appointment.) These key fields correspond to the name of a file in our model. Changing the key fields is like moving the file; changing information in other fields is like changing the contents of the file.

To describe Intellisync in our framework, we just need to generalize the notion of filesystem paths to include names for individual records within files by allowing combinations of key-field values as filename components (e.g., $p = usr.bcp.phonebook.\{lastname{=}Smith,firstname{=}John\}$). The behavior described in the Intellisync manual then follows our specification quite closely. In fact, if we consider the operation of Intellisync just on a single database, then we may drop the clauses of our specification that deal with directories and describe its behavior even more succinctly:

$$\begin{aligned} &\neg dirty_A(p) \\ &\quad \Longrightarrow \quad C(p) = D(p) = B(p) \\ &\neg dirty_B(p) \\ &\quad \Longrightarrow \quad C(p) = D(p) = A(p) \\ &dirty_A(p) \wedge dirty_B(p) \\ &\quad \Longrightarrow \quad C(p) = A(p) \wedge D(p) = B(p) \end{aligned}$$

6.6 Version Control Systems

Another class of systems with some striking similarities to file synchronizers is *version control* or *source control* systems like CVS. Such systems include numerous features (version

histories, alternative branches, etc.) that fall outside the scope of our specification, but their core behavior includes commands like "check in all changes in this group of files, except in cases where the changes conflict with changes that have already been checked in by another project member." Our requirements might be a useful starting point for full specifications of such systems.

7 Extensions

We close by sketching some extensions of our framework.

7.1 Partially Successful Synchronization

If it recognizes conflicting updates, the synchronizer may halt without having made the filesystems identical. Then, the next time the synchronizer runs, there will not be one original filesystem, but two. In general, particular regions of the filesystem may have been successfully synchronized at different times. We can easily refine our specification to handle this case. (Our implementation also handles this refinement.)

Instead of assuming that the replicas had some common state O at the end of the previous synchronization, we introduce into the specification a new filesystem Γ, which records the contents of each path p at the last time when p was successfully synchronized.

The specification of the update detector remains the same as before, except that the *dirty* predicate is defined with respect to Γ. That is, $dirty_S(p)$ must be *true* whenever p refers in S to something different from what it referred to at the end of the last successful synchronization of p.

The reconciler is now extended with an additional output parameter: besides calculating the new states C and D of the two replicas, it returns a new filesystem Γ', which will be used as the Γ input to the next round of synchronization. For each path p, $\Delta(p)$ records the contents of p at the last point where p was successfully synchronized. Formally, we say that the triple (C, D, Γ') is said to be a *synchronization* of a pair of original filesystems (A, B) with respect to predicates $dirty_A$ and $dirty_B$ and original state Γ if, for each relevant path p in (A, B), the following conditions are satisfied:

$$
\begin{array}{l}
\neg dirty_A(p) \\
\quad \Longrightarrow \quad C(p) = D(p) = B(p) = \Gamma'(p) \\
\neg dirty_B(p) \\
\quad \Longrightarrow \quad C(p) = D(p) = A(p) = \Gamma'(p) \\
isdir_{A,B}(p) \\
\quad \Longrightarrow \quad isdir_{C,D}(p) \ \wedge \ isdir_{\Gamma'}(p) \\
dirty_A(p) \wedge dirty_B(p) \wedge \neg isdir_{A,B}(p) \\
\quad \Longrightarrow \quad C(p) = A(p) \ \wedge \ D(p) = B(p) \\
\qquad \wedge \text{ if } A(p) = B(p) \text{ then } \Gamma'(p) = A(p) \\
\qquad\qquad\qquad\qquad \text{else } \Gamma'(p) = \Gamma(p)
\end{array}
$$

7.2 Multiple Replicas

In general, one may wish to synchronize several replicas on different hosts, not just two. We can generalize our requirements specification to handle multiple replicas in a fairly straightforward way.

Let $Id = \{1, 2, \ldots, n\}$ be a set of tags identifying the n replicas to be synchronized. Let the set of original replicas to be synchronized be denoted by $\mathcal{F}_S = \{S_i \mid i \in Id\}$. For any path p, let $D_{p,S}$ be the set of identifiers of replicas that are dirty at p—i.e., $D_{p,S} = \{i \mid dirty@_{S_i}(p)\}$. A set of new replicas $\mathcal{F}_R = \{R_i \mid i \in Id\}$ is said to be a *synchronization* of $\mathcal{F}_S$ with respect to dirtiness predicates $dirty@_{S_i}$ if, for each relevant path p in $\mathcal{F}_S$, the following conditions are satisfied:

$$
\begin{array}{l}
D_{p,S} = \emptyset \\
\quad \Longrightarrow \quad \forall i \in Id.\ R_i(p) = S_i(p) \\
D_{p,S} \neq \emptyset \ \wedge \ \forall i, j \in D_{p,S}.\ S_i(p) = S_j(p) \\
\quad \Longrightarrow \quad \exists j \in D_{p,S}.\ \forall i \in Id.\ R_i(p) = S_j(p) \\
isdir_S(p) \\
\quad \Longrightarrow \quad isdir_R(p) \\
\exists i, j \in D_{p,S}.\ S_i(p) \neq S_j(p) \ \wedge \ \neg isdir_S(p) \\
\quad \Longrightarrow \quad \forall i \in Id.\ R_i(p) = S_i(p)
\end{array}
$$

It is interesting to note that Coda's reconciliation strategy depends on a similar requirement. Coda has a certification mechanism which ensures that reconciliation is safe to proceed. Kumar [Kum94, pages 58-61] proves that, if certification succeeds at all servers, then for each data item d, either (i) d is not modified in any partition, (ii) the final value of d in each partition is equal to the pre-partition value, or (iii) d was modified in exactly one partition.

In a multi-replica system, the process of reconciliation may in general only involve a subset of the replicas at one time. To describe the intended behavior in this case, we would need to combine the above specification with the refinement described in Section 7.1.

7.3 Additional Filesystem Properties

A related generalization offers a natural means of extending our simple model of the filesystem to include properties like read/write/execute permissions, timestamps, type information, symbolic links, etc. For example, a symbolic link can be regarded as a special kind of file whose contents is the target of the link. Similarly, to handle permission bits for files, we take the contents of the file to include both its proper contents and the permission bits.

Hard links are somewhat more difficult to handle, especially if it is possible to create a hard link from inside a synchronized filesystem to some unsynchronized file. However, if this case is excluded, it seems reasonable to handle hard links by annotating each filesystem with a relation describing which files are hard-linked together and taking this additional information into account in the update detector and reconciler.

Acknowledgments

Marat Fairuzov provided a motivating spark for this work by pointing out some of the subtleties of update detection. Luc Maranget and Peter Reiher gave us the benefit of their own deep experience with writing synchronizers. Jay Kistler and Brian Noble helped explore connections with distributed filesystems and gave us many leads and pointers into the literature in that area. Susan Davidson pointed out useful connections with problems in distributed databases, and Ram Venkatapathy advised us on the mysteries of Windows. Brian Smith contributed his usual boundless enthusiasm and helped us begin to see what it would mean to *really* understand synchronization (in the philosophical sense). Conversations with Peter Buneman, Giorgio Ghelli, Carl Gunter, Bob Harper, Michael Levin, Scott Nettles, and Nik Swoboda helped us improve our presentation of the material. Haruo Hosoya, Michael Levin, Jonathan Sobel, and the MobiCom

referees gave us useful comments on earlier drafts of this paper. This work was supported by Indiana University and by NSF grant CCR-9701826.

References

[Bri98] Microsoft Windows 95: Vision for mobile computing, 1998. http://www.microsoft.com/windows95/info/w95mobile.htm.

[Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed databases. *ACM Transactions on Database Systems*, 9(3), Sep. 1984.

[DDD+94] D. Daniels, L. B. Doo, A. Downing, C. Elsbernd, G. Hallmark, S. Jain, Bob Jenkins, P. Lim, G. Smith, B. Souder, and J. Stamos. Oracle's symmetric replication technology and implications for application design. In *Proceedings of SIGMOD Conference*, 1994.

[DGMS85] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3), September 1985.

[DPS+94] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California*, December 1994.

[FM82] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[GPJ93] R. G. Guy, G. J. Popek, and T. W. Page Jr. Consistency algorithms for optimisic replication. In *Proceedings of the First International Conference on Network Protocols*, October 1993.

[HH95] L. B. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location Independent Computing*, Spring 1995.

[Kis96] James Jay Kistler. *Disconnected Operation in a Distributed File System.* PhD thesis, Carnegie Mellon University, 1996.

[Kum94] Puneet Kumar. *Mitigating the effects of Optimistic Replication in a Distributed File System.* PhD thesis, Carnegie Mellon University, December 1994.

[MT94] Murray S. Mazer and Joseph J. Tardo. A client-side-only approach to disconnected file access. In *Workshop on Mobile Computing Systems and Applications*, December 1994.

[PJG+97] T. W. Page, Jr., R. G.. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), December 1997.

[Pow98] PowerMerge software (Leader Technologies), 1998. http://www.leadertech.com/merge.htm.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, October 1997.

[Puma] Designing effective synchronization solutions: A White Paper on Synchronization from Puma Technology. http://www.pumatech.com/syncwp.html.

[Pumb] A white paper on DSX^{tm} Technology – Data Synchronization Extensions from Puma Technology. http://www.pumatech.com/dsxwp.html.

[Rei97] Peter Reiher. Rumor 1.0 User's Manual., 1997. http://fmg-www.cs.ucla.edu/rumor.

[RHR+94] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, June 1994.

[RPG+96] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner. Peer-to-peer reconciliation based replication for mobile computers. In *European Conference on Object Oriented Programming '96 Second Workshop on Mobility and Replication*, June 1996.

[Sch96] Stu Schwartz. The Briefcase—in brief. *Windows 95 Professional*, May 1996. http://www.cobb.com/w9p/9605/w9p9651.htm.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado*, December 1995.